

WebSpeed Developer's Guide

© 2001 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

.

May 2001



Product Code: 4417
Item Number: 81119;3.1C

Contents

Preface . . . . . **xiii**
Purpose xiii
Audience xiii
Organization of This Manual xiv
How to Use This Manual xv
Typographical Conventions xvi
Syntax Notation xvii
WebSpeed Messages xxi
Other Useful Documentation xxii
Getting Started xxii
Development Tools xxiii
Reporting Tools xxiv
4GL xxv
Database xxvi
DataServers xxvi
SQL-89/Open Access xxvi
SQL-92 xxvii
Deployment xxvii
WebSpeed xxviii
Reference xxviii

1. Introducing WebSpeed **1–1**
1.1 What is WebSpeed? 1–2
1.2 Advantages of WebSpeed 1–2
1.3 An Overview of WebSpeed 1–3
1.4 SpeedScript 1–5

2. Overview of WebSpeed Tools **2–1**
2.1 The AppBuilder 2–2

2.1.1 Visual and Nonvisual Objects 2–2
2.1.2 SmartDataObjects 2–2
2.1.3 Templates and Wizards 2–3
2.1.4 Editors 2–4
2.2 Running the WebSpeed AppBuilder 2–4
2.2.1 Preliminary Requirements for Running the WebSpeed AppBuilder 2–4
2.2.2 Starting the WebSpeed AppBuilder 2–5
2.2.3 Connecting to a Database Server 2–5
2.2.4 Specifying a Default Browser 2–6
2.2.5 Specifying a WebSpeed Broker 2–6
2.2.6 Setting the Development Mode 2–7
2.2.7 AppBuilder Documentation 2–8
2.3 WebTools 2–9
2.3.1 WebTools Documentation 2–10
2.4 Other WebSpeed Components 2–11
2.5 WebSpeed Source Files 2–12

3. WebSpeed Architecture 3–1
3.1 WebSpeed Components 3–2
3.1.1 WebSpeed Component Distribution 3–3
3.1.2 WebSpeed Transaction Server Products 3–4
3.2 How WebSpeed Handles a Request 3–5
3.3 WebSpeed Configuration and Management 3–11
3.3.1 Progress Explorer 3–11
3.3.2 Command-line Utilities 3–11
3.3.3 SpeedStart 3–11

4. Sample Web Objects 4–1
4.1 Types of Web Objects 4–2
4.1.1 Standard or Static HTML Web Objects 4–2
4.1.2 Embedded SpeedScript Web Objects 4–2
4.1.3 CGI Wrapper Web Objects 4–3
4.1.4 HTML Mapped Web Objects 4–3
4.1.5 General Criteria for Use 4–4
4.2 Using Web Object Examples 4–5
4.2.1 Web Object URLs and PROPATH 4–6
4.3 Embedded SpeedScript Examples 4–7
4.3.1 A Simple Query 4–8
4.3.2 Handling Form Input 4–10
4.4 CGI Wrapper Example 4–13
4.5 HTML Mapping Examples 4–15
4.5.1 Simple HTML Mapping 4–16
4.5.2 Complex HTML Mapping 4–22
4.5.3 Complex HTML Mapping That Includes a SmartDataObject . . 4–31

5. SpeedScript 5–1
5.1 SpeedScript and Progress 4GL 5–2
5.1.1 Similarities 5–2
5.1.2 Differences 5–2
5.2 Elements of SpeedScript Syntax 5–4
5.2.1 Running Procedures and Web Objects 5–6
5.2.2 Blocks and Context 5–7
5.2.3 Block Context and Resource Scope 5–8
5.2.4 Compile-time Versus Run-time Code 5–10
5.3 WebSpeed Preprocessors 5–13
5.4 WebSpeed API Functions 5–14
5.4.1 Message Handling 5–14
5.4.2 Passing Information Between Web Requests 5–16
5.4.3 General Information Exchange 5–21
5.4.4 Managing Date and Time Information 5–22
5.4.5 Generating Web Page Headers 5–23
5.4.6 Checking Configuration Options 5–24
5.5 XML Support in SpeedScript 5–24
5.6 Overview of Embedded SpeedScript 5–25
5.7 Authoring Embedded SpeedScript Files 5–26
5.8 How to Embed SpeedScript in HTML 5–27
5.8.1 Statement Escapes 5–27
5.8.2 Expression Escapes 5–29
5.9 Building Interactive Web Pages with Embedded SpeedScript 5–30
5.9.1 Passing Parameters 5–30
5.9.2 Managing Dynamic Pages with Forms 5–31
5.10 Using <META> and <!--WSMETA --> Tags 5–36
5.10.1 Specifying File Type Options 5–37
5.10.2 Specifying HTTP Header Information 5–40
5.10.3 Generating Information Prior to HTTP Header Output 5–41
5.11 Developing Embedded SpeedScript Web Objects 5–41
5.11.1 Building Embedded SpeedScript Web Objects 5–41
5.11.2 Running Embedded SpeedScript Web Objects 5–42

6. Handling DISPLAY Output 6–1
6.1 Working with DISPLAY Output 6–2
6.1.1 Directing DISPLAY Output 6–2
6.1.2 Formatting DISPLAY Output 6–4
6.2 Changing Output Format Defaults 6–8
6.2.1 Setting {&WEBSTREAM} Defaults 6–8
6.2.2 Setting HTML Attribute Defaults 6–8

7. Generating HTML Visualizations 7–1
7.1 HTML Visualizations 7–2

7.2 Using the Support for HTML Table Visualizations 7–2
7.3 Generating HTML Tables with a Custom Tag 7–2
7.3.1 Rules for Using Custom Tags 7–3
7.3.2 Using the <!--WSTAG--> Custom Tag 7–3
7.3.3 How tagrun.p Interacts with the Custom Tag 7–4
7.3.4 Using the Table Template 7–5
7.3.5 Compiling the Table Template 7–5
7.4 Generating HTML Tables Directly from SpeedScript 7–6
7.5 Adding a Template to AppBuilder 7–7
7.6 Support for Other HTML Visualizations 7–8

8. Controlling WebSpeed Transactions 8–1
8.1 Defining State 8–2
8.1.1 Web Object States 8–2
8.1.2 WebSpeed Application States 8–3
8.1.3 Advantages and Drawbacks of State-persistence 8–3
8.2 Understanding WebSpeed Transaction Control 8–4
8.2.1 How To Make a Web Object State-aware 8–5
8.2.2 Web Objects in Stateless and State-persistent Contexts 8–7
8.3 Primary and Secondary Web Objects 8–8
8.3.1 Running Primary and Secondary Web Objects. 8–8
8.3.2 State-aware Web Objects and Persistence 8–10
8.4 Implementing Transaction Control 8–11
8.4.1 Modifying the outputHeader Procedure 8–11
8.4.2 Modifying the process-web-request Procedure 8–13
8.4.3 Overriding Procedures and Functions. 8–13
8.4.4 Modifying the Attribute List 8–14
8.4.5 Modifying the User Field List 8–15
8.5 Transaction Control with Embedded SpeedScript Web Objects 8–16
8.6 Transaction Control with CGI Wrapper Web Objects 8–16
8.7 Transaction Control with HTML-mapping Web Objects 8–17
8.7.1 Anatomy of process-web-request in HTML-mapping 8–18
8.7.2 Modifying the Request Logic. 8–20
8.7.3 Moving Data Through the HTML Mapping Web Object 8–21
8.7.4 Creating Definitions 8–25
8.7.5 Modifying the Main Code Block. 8–26
8.7.6 Customizing Field Object Control Handlers 8–28
8.8 Handling State-aware Time-outs 8–33
8.8.1 Using a Web Object Time-out Handler 8–34
8.8.2 Resetting a Web Object's Time-out Period 8–35
8.8.3 Allowing WebSpeed to Handle Time-outs. 8–36

9. Controlling Database Transactions 9–1
9.1 What Is a Database Transaction? 9–2

9.1.1 DB Transactions in Code 9–2
9.1.2 All-or-nothing Processing 9–4
9.2 Understanding the Scope of DB Transactions 9–5
9.2.1 Subtransactions 9–7
9.2.2 Controlling Where DB Transactions Begin and End 9–8
9.3 SpeedScript Components and DB Transactions 9–9
9.3.1 Subprocedures 9–10
9.3.2 File Input 9–10
9.3.3 Program Variables 9–11
9.4 DB Transactions in Applications 9–11
9.4.1 Multi-database Applications 9–11
9.4.2 Distributed Applications 9–13
9.5 Determining When DB Transactions Are Active 9–13
9.6 Transaction System Mechanics 9–13
9.6.1 Database Transaction Mechanics 9–13
9.6.2 Subtransaction Mechanics 9–14
9.7 Efficient DB Transaction Processing 9–14
9.8 Multi-page DB Transactions 9–15
9.8.1 Managing Multi-page DB Transactions 9–15
9.8.2 Working with the Multi-page Transaction Example 9–16
9.8.3 Guidelines for Usage 9–18

10. Running Applications 10–1
10.1 Setting Up an Application Environment 10–2
10.1.1 Creating a Working Application Directory 10–2
10.1.2 Moving Application Files to Appropriate Directories 10–3
10.1.3 Compiling Web Objects 10–4
10.1.4 Making Your Application Secure 10–5
10.2 Starting and Stopping an Application 10–5
10.3 Debugging WebSpeed Applications 10–6
10.3.1 Adding debug=on to URL 10–6
10.3.2 Persistent Debugging Uses Cookies 10–6
10.3.3 Calling the Virtual Debug Web Object 10–7
10.3.4 Debugging and Administrative Options 10–7
10.3.5 Utilizing Debugging in Your Application 10–9
10.3.6 Reading the Broker's Log Files 10–10

Index Index–1

Figures

Figure 1–1: Components of the WebSpeed Development Environment 1–4
Figure 2–1: The AppBuilder Main Window 2–5
Figure 2–2: The Development Mode Button 2–8
Figure 2–3: WebSpeed WebTools Main Page 2–9
Figure 4–1: Running w-sstcst 4–9
Figure 4–2: User Input Page Generated by w-sstget 4–11
Figure 4–3: Customer Information Generated by w-sstget 4–12
Figure 4–4: Tree View of HTML Mapping Source 4–18
Figure 4–5: Web Page Generated by w-cstget 4–20
Figure 4–6: Web Page Generated from w-cstinf.w 4–22
Figure 4–7: Web Page Generated from w-custdir.w 4–31
Figure 4–8: Tree View of an Untitled HTML-mapping Object 4–35
Figure 8–1: A Web Object Running Stateless and State-persistent 8–7
Figure 8–2: Primary and Secondary Web Objects in a WebSpeed Transaction . . . 8–9
Figure 8–3: Standard HTML-mapping process-web-request Model 8–19
Figure 8–4: Data Flow Through an HTML-mapping Web Object 8–21
Figure 9–1: Database Transaction Definition 9–2
Figure 9–2: Database Transaction Undo Processing 9–3
Figure 9–3: Database Transaction Involving Two Tables 9–4
Figure 9–4: Database Transaction Scope 9–5
Figure 9–5: Multiple DB Transactions in a Procedure 9–6
Figure 9–6: DB Transactions and Subprocedures 9–10
Figure 9–7: WebSpeed Multi-page Transaction Model 9–19

Tables

Table 2–1:	WebTools	2–10
Table 3–1:	Maximum WebSpeed Component Distribution	3–3
Table 3–2:	WebSpeed Products	3–4
Table 5–1:	WebSpeed Preprocessors	5–13
Table 5–2:	Message API Functions	5–14
Table 5–3:	Cookie-passing API Functions	5–17
Table 5–4:	URL API Functions	5–18
Table 5–5:	Hidden Field API Functions	5–19
Table 5–6:	User Field API Functions	5–20
Table 5–7:	General Information Exchange API Functions	5–21
Table 5–8:	Date and Time API Functions	5–22
Table 5–9:	Web Page Header API Functions	5–23
Table 5–10:	Configuration API Functions	5–24
Table 5–11:	Embedded SpeedScript Statement Escapes	5–27
Table 5–12:	Embedded SpeedScript Expression Escapes	5–29
Table 6–1:	WEB-CONTEXT Handle HTML Attributes	6–6
Table 9–1:	Starting Database Transactions and Subtransactions	9–8

Procedures

w-sstcst.html 4–8
w-sstget.html 4–10
w-forcst.w 4–13
w-cstget.htm 4–16
w-cstget.off 4–17
w-cstget.w 4–19
w-custinf.htm 4–24
w.cstinf.w (RUN outputHEADER) 4–25
w-cstinfo.w (GET branch) 4–25
w-cstinf.w (POST branch) 4–26
findCustomer 4–28
w-custdir.htm 4–34
Web.input 4–36
inputFields 4–37
outputFields 4–38
process-web-request 4–39
sample1 5–4
sample2 5–5
sample3 5–7
sample4 5–9
sample5 5–12
sample6 5–12
escript1.htm 5–25
escript2.htm 5–28
escript3.htm 5–29
escript4.htm 5–30
browse.html 5–31
browse.html 5–32
browse.html 5–33
browse.html 5–34
browse.html 5–35
meta.html 5–39
web.cst 7–7
w-cstinf.w 8–25
tagmap.dat 8–28
ncust-wo.htm 8–30
ncust-wo.off 8–31
web.input 8–32
web.output 8–32
sample7 9–12

Preface

Purpose

WebSpeed represents a significant advance in the efforts to bring commerce to the Web. It allows you to build applications that support full business transactions directly between any Web user and a place of business.

This manual aims to provide a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

Audience

This manual assumes that you are an experienced Webmaster and computer programmer. You should be thoroughly familiar with HyperText Markup Language (HTML), HyperText Transfer Protocol (HTTP), and the Common Gateway Interface (CGI) programming used for producing and transferring documents over the Web. Fluency with block-structured languages such as Pascal, C/C++, Visual Basic, or Progress and a working familiarity with object-oriented and event-driven programming is helpful but not necessary.

Organization of This Manual

Chapter 1, "Introducing WebSpeed"

Describes the components and capabilities of WebSpeed.

Chapter 2, "Overview of WebSpeed Tools"

Describes the tools and utilities used in WebSpeed application development.

Chapter 3, "WebSpeed Architecture"

Describes the WebSpeed components and how they work together to execute a WebSpeed application.

Chapter 4, "Sample Web Objects"

Describes several simple Web Objects (which are components of WebSpeed applications) that illustrate some basic WebSpeed techniques and concepts.

Chapter 5, "SpeedScript"

Describes the SpeedScript programming language and how to build Web objects with embedded SpeedScript.

Chapter 6, "Handling DISPLAY Output"

Describes how to use and modify the default formatting for Web object output generated by the DISPLAY statement.

Chapter 7, "Generating HTML Visualizations"

Describes how to generate HTML visualizations, using the installed custom tag and template for generating HTML tables as an example.

Chapter 8, "Controlling WebSpeed Transactions"

Describes Web object states and how to implement them with the different types of WebSpeed Web objects.

Chapter 9, "Controlling Database Transactions"

Describes how SpeedScript controls Progress database transactions and how you can use state-aware Web objects to control multi-page database transactions on the Web.

Chapter 10, "Running Applications"

Provides basic information on configuring the WebSpeed Transaction Server to test and deploy WebSpeed applications, including information on WebSpeed debugging techniques.

Index

How to Use This Manual

To run the example Web objects mentioned in this manual, do the following from the AppBuilder:

1 ♦ Select Tools→ Database Connections and connect to the sample sports2000 database.

2 ♦ Select Options→ Preferences, click on the WebSpeed tab, and specify your Web browser and your Broker URL.

3 ♦ Select Tools→ WebTools and then select File Tools from the left frame of the browser window.

4 ♦ From the File Tools interface navigate to *install-path*`/src/web/examples`. You will see the example Web objects in a file list and you can view or run them.

NOTE: Avoid accidentally overwriting the example files. If you want to edit or compile any of the files in the examples directory, be sure to make a working copy of the examples directory.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

  ```
  FOR EACH Customer:
     DISPLAY Name.
  END.
  ```

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

  SYNTAX

  ```
  DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
  ```

  In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

SYNTAX

```
INITIAL [ constant [ , constant ] ... ]
```

NOTE: The ellipsis (**...**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

  ```
  { BY expression [ DESCENDING ] }
  ```

  In some cases, braces are not a syntax notation, but part of the language.

  For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

  SYNTAX

  ```
  { &argument-name }
  ```

- A vertical bar (**|**) indicates a choice.

  In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

  SYNTAX

  ```
  PRESELECT [ EACH | FIRST | LAST ] record-phrase
  ```

In this example, you must select one of *logical-name* or *alias*:

SYNTAX

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

  ```
  MAXIMUM ( expression , expression [ , expression ] ... )
  ```

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

  ```
  MESSAGE { expression | SKIP [ (n) ] } ...
  ```

  In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

  SYNTAX

  ```
  { include-file
      [ argument | &argument-name = "argument-value" ] ... }
  ```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

In this example, WITH is followed by several optional items:

SYNTAX

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

SYNTAX

```
ASSIGN {   { [ FRAME frame ]
               { field [ = expression ] }
               [ WHEN expression ]
           } ...
         | { record [ EXCEPT field ... ] }
       }
```

WebSpeed Messages

WebSpeed displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while WebSpeed is running a procedure (for example, if it cannot find a record with a specified index field value).
- Compile messages inform you of errors found while WebSpeed is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while WebSpeed is getting ready to execute (for example, if you entered an invalid startup parameter).

WebSpeed messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

Use the WebTools Message tool to see more information about WebSpeed messages. In addition, the online help for many Progress tools includes the following Help menu options:

- Choose Help→ Recent Messages to display detailed descriptions of the most recent WebSpeed message and all other messages returned in the current session.
- Choose Help→ Messages, then enter the message number to display a description of any WebSpeed message. (If you encounter an error that terminates WebSpeed, make a note of the message number before restarting.)

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Version 3 Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed! (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

1

Introducing WebSpeed

This chapter is an overview of the capabilities and components of WebSpeed. It includes the following topics:

- An explanation of what WebSpeed is
- A list of the advantages of WebSpeed
- An overview of the WebSpeed development environment
- An introduction to the SpeedScript programming language

1.1 What is WebSpeed?

WebSpeed is a development and deployment environment that enables you to build robust, data-driven Internet Transaction Processing (ITP) applications for the World Wide Web (WWW). WebSpeed provides database connectivity and transaction management to browser-based applications. In addition, you can use WebSpeed to web-enable existing Progress applications.

If you want to see a few sample WebSpeed applications, see *Getting Started with WebSpeed*. That manual contains instructions on how to run the WebSpeed sample applications and it also contains a tutorial on how to create a WebSpeed application.

1.2 Advantages of WebSpeed

The WebSpeed architecture provides the following advantages to your ITP application development process:

- Native database support includes access to Progress, ORACLE, and DB2/400, plus ODBC access to DB2, MS SQL Server, Sybase, MSAccess, Informix, and other data sources.
- Compatible with any ISAPI, NSAPI, or CGI-compliant Web servers.
- Wizards and templates can be used to create professional, multi-page ITP applications with little or no coding, significantly cutting development time while preserving the ability to customize your application.
- Flexible programming model lets you structure development based on your project requirements, embedding SpeedScript directly into your HTML pages or using HTML mapping to bind HTML files to business logic.
- Scripting Lab enables you to test code fragments quickly before adding them to your WebSpeed application, increasing productivity and helping you create feature-rich user interfaces and browser-side validation routines using JavaScript, VBScript, and Java applets.
- Open architecture integrates with your choice of security solutions, including firewall, authentication, and encryption technologies.
- High-performance, scalable architecture delivers sub-second transaction processing and handles thousands of simultaneous users under peak load.

- Dynamic load balancing ensures high availability of transaction processing resources in a distributed, multi-tier environment.
- Flexible state management offers full support for extended database queries and updates using stateless, state-aware, or state-persistent Web objects.

1.3 An Overview of WebSpeed

The WebSpeed development and deployment environment consists of four major software components:

- **The WebSpeed Workshop**, which includes the following:
  - **AppBuilder**, a graphical tool that contains wizards, templates, compilers, and editors for building Web Objects. The AppBuilder is also a launching point for various database administration tools (the Data Dictionary, for example).
  - **WebTools**, a set of browser-based utilities.
  - **WebSpeed Development Server**, actually a WebSpeed Transaction Server that responds to requests, executes WebSpeed applications (also known as Web objects), and delivers HTML pages to the Web Server.
- **A Web Browser**, the client software where the request for WebSpeed services originates. You use a browser to test and run your WebSpeed applications. You also use a browser to run WebTools.
- **A Web Server**, the software that passes requests between a browser and the WebSpeed components. The Web Server receives requests for WebSpeed services and it passes the request to a the WebSpeed Development Server. When the request has been processed by WebSpeed, the Web server returns an HTML page back to the browser.
- **A Data Source**, a Progress database server or any of a wide variety of other database servers. You can also use a SmartDataObject as a data source. See the *Progress AppBuilder Developer's Guide* for information on creating SmartDataObjects.

NOTE: Only the first item, WebSpeed Workshop, is included in the WebSpeed product. However, depending on your licensing option, you might also have a Progress data source as well.

For more information about WebSpeed components and their distribution in a network environment, see Chapter 3, "WebSpeed Architecture." Also see the *WebSpeed Installation and Configuration Guide* for more information on configuring WebSpeed and on system requirements.

Figure 1–1 shows the relationship between the components in the WebSpeed development environment.

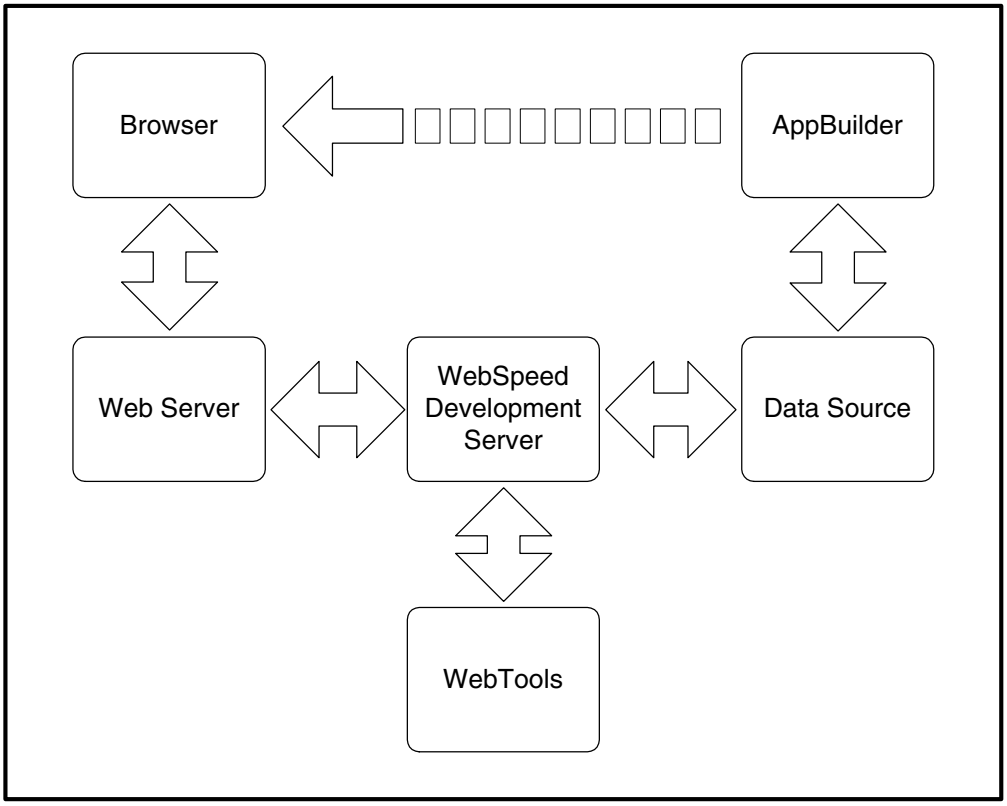


Figure 1–1: Components of the WebSpeed Development Environment

Basically, you use WebSpeed AppBuilder connected to a data source to create Web objects. A Web object may, for example, perform a query on the data source. When you test or run Web objects, the AppBuilder spawns a browser. The broken line in the diagram indicates that there is no sustained connection between the browser and the AppBuilder. The browser passes the request to the Web Server and the Web Server sends the request to the WebSpeed Development Server. The WebSpeed Development Server, which is also connected to the data source, executes the Web object. The Web object formats its results as an HTML page. The WebSpeed Development Server passes the HTML page back to the browser through the Web Server.

You can start the WebTools utilities directly from a browser by supplying a URL. Or, you can start WebTools from the AppBuilder tools menu. In the later case, the AppBuilder merely spawns a browser. The browser then generates a request to the Web Server, which passes it to the WebSpeed Development Server for execution. The Development Server generates the Web page that is returned to the browser through the Web server.

All the components of the WebSpeed Development Environment can reside on a single machine, or on multiple machines in a network environment. However, there are some restrictions on a network deployment of WebSpeed. For example, the WebSpeed Messenger, a component of the WebSpeed Development Server, must reside on the same machine as the Web Server. For more information about requirements for running WebSpeed, see the *WebSpeed Installation and Configuration Guide*.

For more information about WebSpeed AppBuilder and WebTools, see Chapter 2, "Overview of WebSpeed Tools."

For more information about the WebSpeed architecture, see Chapter 3, "WebSpeed Architecture."

1.4 SpeedScript

Another component of the WebSpeed development environment is the support of SpeedScript for Web application development.

SpeedScript is an interpreted, block-structured, and statement-oriented programming language based on Progress 4GL. It provides native access to the Progress RDBMS and DataServers, as well as to Web servers for Web page input and output.

SpeedScript enables you to quickly prototype, develop, and maintain re-usable application business logic. It also includes object-oriented features like super procedures and dynamic queries and buffers.

Unlike SQL, to which it is sometimes compared, SpeedScript constitutes a complete application programming language, with equally powerful facilities for managing business logic and database manipulation. At the same time, SpeedScript includes an implementation of SQL-89 if you prefer to access databases this way.

Depending on your application and the type of Web objects you are building, the AppBuilder can automatically generate some or all of the SpeedScript used to build your Web objects. For more information about SpeedScript, see Chapter 5, "SpeedScript."

NOTE: This manual aims to provide a basic knowledge of SpeedScript, to get you started. For more information on the elements of the language, see the *Progress Programming Handbook*.

2

Overview of WebSpeed Tools

The WebSpeed development environment features fast-prototyping tools and components to generate dynamic Web pages as quickly and easily as possible. It does not necessarily replace all of your existing Web development environment, but adds capabilities that can increase development productivity as well as the depth and scope of your applications.

This chapter introduces you to the tools and utilities used in WebSpeed application development, including:

- The AppBuilder
- Running the WebSpeed AppBuilder
- WebTools
- Other WebSpeed Components
- WebSpeed Source Files

2.1 The AppBuilder

The AppBuilder is a multi-purpose application development environment. You can use it as a visual programming environment to create character- or GUI-based client/server applications. In addition, you can use the AppBuilder for WebSpeed to create HTML-based Web applications.

The AppBuilder is installed with one of the following licensing options:

- WebSpeed AppBuilder, which supports creating, testing, and deploying Web objects
- ProVision AppBuilder, which supports creating, testing, and deploying traditional n-tier client/server applications (either character- or GUI-based)
- Provision Plus AppBuilder, which supports both client/server and Web application development, plus integration with other non-Progress applications such as Java or ActiveX

This manual focuses on running the WebSpeed AppBuilder. For more information about the ProVision AppBuilder and the ProVision Plus AppBuilder, see the *Progress AppBuilder Developer's Guide*.

2.1.1 Visual and Nonvisual Objects

If you are using the ProVision Plus AppBuilder licensing option, a separate window called the Object Palette appears when you open the AppBuilder. If you are using the WebSpeed AppBuilder licensing option, the Object Palette does not appear. The Object Palette allows you to create visual objects (buttons, for example).

In WebSpeed, you only create nonvisual objects, which include embedded SpeedScript, CGI Wrapper, and HTML-mapping Web objects. (SpeedScript is an interpreted, block-structured, and statement-oriented language based on Progress 4GL.) You cannot manipulate nonvisual objects graphically at design time, and they do not have a visualization element at run time.

2.1.2 SmartDataObjects

WebSpeed AppBuilder supports a nonvisual SmartObject called the SmartDataObject. Written exclusively in the Progress 4GL and SpeedScript compatible, the SmartDataObject allows you to define and manage business logic and data verification. See the *Progress AppBuilder Developer's Guide* for more information about SmartDataObjects.

2.1.3 Templates and Wizards

WebSpeed AppBuilder provides templates and wizards to assist you when you create new Web objects, HTML files, and SmartDataObjects. From the main window of the AppBuilder, choose File→ New to see the list of templates and wizards that are available.

If you have a WebSpeed license, the Objects list displays only WebSpeed objects. If you have a license for the Unified AppBuilder product, the list also displays objects that you can use for Progress applications.

You can use the WebSpeed wizards to create Web objects without writing any HTML or SpeedScript code. The WebSpeed wizards prompt you for the necessary information and automatically generate the required code.

The WebSpeed wizards are:

- **Report**—Creates a tabular view of the database fields you specify. You can add navigation buttons for displaying the first, previous, next, and last set of records. You can also include a text entry field to allow the user to enter search criteria.
- **Detail**—Creates an updatable form to display database data, based on the selection criteria you specify. You can add transaction control buttons to allow users to add or delete records and to submit or reset changes.
- **HTML Mapping**—Maps form fields in an existing HTML file to database fields.

In addition to the wizards, the following templates are available:

- **Main**—A Welcome or a static text page for a WebSpeed application.
- **CGI Wrapper**—A Web object that dynamically generates HTML within the context of SpeedScript.
- **Blank**—A basic HTML file that includes an embedded SpeedScript section.
- **Frameset**—An HTML file that creates three frames (a banner and two columns). The template also contains markup that displays a message when the browser does not support frames.
- **Report**—An HTML file with preprocessor definitions for creating a tabular view of database fields with navigation buttons.
- **Table**—A file for formatting database results into an HTML table. This is not a stand-alone file since it does not contain all the required HTML tags. It is meant to be inserted into another HTML file.

2.1.4 Editors

The Code Section Editor is the AppBuilder's main tool for editing procedure files. The Code Section Editor is integrated with the AppBuilder, unlike other editors. As a result, when you invoke the Section Editor, it displays an appropriate code section. The Section Editor also preserves the procedure file's format by presenting only some sections of the file.

To invoke the Section Editor, choose the Code button or Window→ Code Section Editor from the AppBuilder main menu. You can also double-click on an object to invoke the Section Editor, if that feature is enabled in the Options→ Preferences window.

You can also edit procedure files with conventional text editors, such as NotePad.

2.2 Running the WebSpeed AppBuilder

This section is an overview that describes how to start the AppBuilder and set it up for WebSpeed development.

2.2.1 Preliminary Requirements for Running the WebSpeed AppBuilder

Before you can use the WebSpeed AppBuilder to develop Web objects, the following must be running and accessible:

- **The AdminServer**, which provides support for configuration utilities and grants access to each instance of an installed Progress product. By default, on Windows NT systems, the AdminServer starts automatically and runs as an NT service.
- **The NameServer**, which manages a pool of brokers including the WebSpeed Broker. The NameServer helps to direct client requests to the correct broker. By default, it starts when the AdminServer starts.
- **A Database Server**, which manages a connection between the AppBuilder and the database. In addition, the database server manages a connection between the WebSpeed Broker and the database.
- **A WebSpeed Broker**, which manages the WebSpeed Agents that execute Web objects. The WebSpeed Broker must be connected to the database server when it starts.
- **A Web Server**, which manages Web requests and returns Web pages to the browser. The Web server must have the WebSpeed Messenger installed in its `\scripts` directory.

See Chapter 3, "WebSpeed Architecture," for a description of how these servers and brokers work together to execute a Web object.

2.2.2 Starting the WebSpeed AppBuilder

Start the AppBuilder from within the Progress Program Group, based on your install path. For example, from the Start menu, choose Programs→ Progress→ AppBuilder. The main window of the AppBuilder opens, as shown in Figure 2–1.



Figure 2–1: The AppBuilder Main Window

In addition, you can start the AppBuilder from the Application Development Environment (ADE) Desktop and any tool that has a Tools menu, like the Data Dictionary or the Procedure Editor. You can also start the AppBuilder directly from an operating system command shell.

For more information about starting the AppBuilder and about the AppBuilder main window, see the *Progress AppBuilder Developer's Guide*.

2.2.3 Connecting to a Database Server

You must connect WebSpeed AppBuilder to a database server before you can begin to develop Web objects. The database must be either the same database that is connected to the WebSpeed Broker, or it must have the same schema.

If the database server is running, do the following:

1 ♦ From the AppBuilder main window, choose Tools→ Database Connections.

2 ♦ Choose the Connect command button.

3 ♦ Choose Options.

4 ♦ Select the Multiple Users check box.

5 ♦ Enter the pathname of the database in the Physical Name text field.

6 ♦ Choose the OK command button.

For more detailed information about starting database servers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*. Also, *Getting Started with WebSpeed* contains a detailed example of starting a database server for the sample Sports2000 database.

2.2.4 Specifying a Default Browser

You need to specify a default browser in order to run and test your WebSpeed applications. Follow these steps:

1 ♦ Select Options→ Preferences from the AppBuilder main window.

2 ♦ Select the WebSpeed tab.

3 ♦ Specify the pathname of the browser that you intend to use during development.

For example, `C:\Program Files\Netscape\Communicator\Program\netscape.exe` and `C:\Program Files\Plus!\Microsoft Internet\iexplore.exe` are typical pathnames for Netscape Navigator and Internet Explorer respectively.

2.2.5 Specifying a WebSpeed Broker

Before you can begin to develop Web objects, you must specify a WebSpeed Broker in the AppBuilder. The WebSpeed Broker must be connected to the same database that is connected to the AppBuilder (or the database must have the same schema as the database that is connected to the AppBuilder).

For information on configuring and starting WebSpeed Brokers, see the *WebSpeed Installation and Configuration Guide*. Also, *Getting Started with WebSpeed* contains a detailed example of starting a WebSpeed Broker connected to the sample Sports2000 database.

To specify a WebSpeed Broker in the AppBuilder, do the following:

1 ♦ Select Options→ Preferences from the AppBuilder main window.

2 ♦ Select the WebSpeed tab.

3 ♦ Enter the URL of the WebSpeed Broker in the Broker URL field.

4 ♦ Choose the Test button to verify the connection. (Be sure that you specify the path of your default browser's executable file before testing.)

5 ♦ Choose the OK command button.

The following shows the general syntax for specifying a Broker URL:

```
http://host_name[:port]/scripts_dir/messenger/WService=broker
```

host_name

Specifies the name of the machine that is running the Web server.

port

Specifies the port number of the Web Server. The port number is optional if the Web Server uses the default port number, which is 80. For example, if a Web Server's port number is 88, the initial part of the URL might be specified as `http://myhost:88`.

scripts_dir

Specifies the Web server scripts directory for a CGI or ISAPI Messenger. Omit this component if you using an NSAPI Messenger.

messenger

Specifies the messenger name. For a CGI Messenger, you add the filename (or pathname relative to the scripts directory) of your CGI Messenger script. For an ISAPI Messenger, you add the filename (or pathname relative to the scripts directory) of the DLL (`wsisa.dll`). For an NSAPI Messenger, you add the filename of the DLL (`wsnsa.dll`).

broker

Specifies the WebSpeed broker name.

Click on the Test button to verify the connection. However, be sure that you specify the path of your default browser's executable file before running a test of the connection.

2.2.6 Setting the Development Mode

The WebSpeed AppBuilder supports two development modes: local and remote.

Use local mode when all of the following conditions are true:

- The WebSpeed AppBuilder is running on the same machine as the WebSpeed Development Server.
- Both the WebSpeed AppBuilder and the WebSpeed Development Server have the same working directory.
- Both the WebSpeed AppBuilder and the WebSpeed Development Server have the same PROPATH settings.

The working directory and PROPATH are set during installation of WebSpeed. For more information, see the *WebSpeed Installation and Configuration Guide*.

Use remote mode when the WebSpeed Development Server is on a remote machine, and when either of the following conditions are true:

- The WebSpeed Development Server and the AppBuilder have different PROPATH settings.
- The WebSpeed Development Server has a different working directory than the AppBuilder. (In remote mode, files are saved in WebSpeed Development Server's working directory on the remote machine.)

To change the development mode, click on the Development Mode toggle button, which is the last button in the AppBuilder tool bar. The Development Mode button in remote and local modes is shown in Figure 2–2.

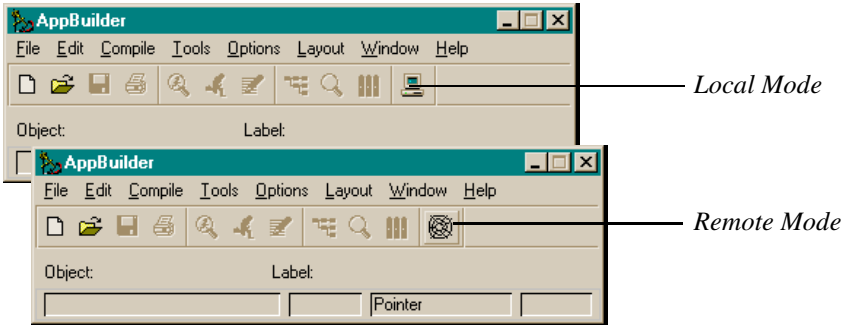


Figure 2–2: The Development Mode Button

NOTE: If you have network or other problems and cannot access a remote WebSpeed Development Server, you can switch to local mode to create and save source files. However, you cannot compile source files that contain Embedded SpeedScript unless remote mode is enabled.

2.2.7 AppBuilder Documentation

Detailed online help for all the AppBuilder features is available from the main Help menu and from help buttons on most dialog boxes. Also see the *Progress AppBuilder Developer's Guide* for more information about using the AppBuilder.

For step-by-step instructions on using WebSpeed AppBuilder to create Web objects, see *Getting Started with WebSpeed*.

2.3 WebTools

WebTools are a collection of browser-based utilities that allow you to perform development tasks and to access information. Each tool is directly accessible by clicking a link from the WebSpeed WebTools page.

The primary advantage of WebTools is that you do not have to install WebSpeed software locally in order to use them. You can access WebTools by specifying the WorkShop URL in your browser (`host_name/scripts/cgiip.exe/WService=wsbroker1/workshop`, for example).

However, if you have WebSpeed installed, you can also access WebTools from the Tools menu in AppBuilder. Select Tools→ WebTools from the AppBuilder main window.

Figure 2–3 shows the WebTools main page.



Figure 2–3: WebSpeed WebTools Main Page

The WebTools main page consists of two frames:

- **Detail frame**—An index of links to tools and to help
- **Main frame**—The work area for the tool you select in the Detail frame

Table 2–1 gives a brief description of each one of the WebTools. For more detailed information, see the online help, which is accessible from the main frame of each WebTool.

Table 2–1: WebTools

Editor	A basic, single-document text editor.
File Tools	A file listing and file filter utility that allows you to view, run, compile, extract, and delete files on the Development Server.
OS Command	An interface for entering operating system commands.
Scripting Lab	An editor and compiler tool for testing small portions of your application. Also contains small code samples.
Databases	Schema information for all databases connected to the WebSpeed Agent.
Agent Variables	A display of current values of CGI, WebSpeed, Web-Context, and Session variables.
ProPath	A list of the files and directories in your WebSpeed Agent's PROPATH, plus access to File Tools when you select a file.
Virtual System Table	Shows the performance statistics that are saved in a database's virtual system tables.
Object State	A listing of all active Web objects showing each object's type and current Web state.
Messages	Access to information about error messages.
Help	Online help for WebTools and for sample applications.

2.3.1 WebTools Documentation

Getting Started with WebSpeed contains a detailed description of each of the WebTools.

From WebSpeed WebTools, you can access the Tools online help system:

- To get instructions for performing WebTools tasks, click the Help link. Then click the How to link. A list of each how-to topic, with a link to instructions for accomplishing a task, appears in a separate browser window.

- To look up a WebSpeed or Internet term, click the Help link. Then click the Glossary link. An indexed list of terms appears in a separate browser window.
- To view a description of each WebTool and access information about how to work with each tool, click the WebSpeed WebTools link.
- To access information about how to use the current tool or option, click the Help icon of any WebSpeed WebTool main frame.

2.4 Other WebSpeed Components

The WebSpeed development environment also contains the following components:

- **SpeedScript**—The implementation language for Web objects. SpeedScript includes a built-in Web server input/output (I/O) system that provides access to the CGI environment and Web page output stream for a Web request. For more information on SpeedScript, see Chapter 5, "SpeedScript." Also see the *Progress Language Reference* and the *Progress Programming Handbook*.
- **WebSpeed global variables**—A rich set of variables available to all Web objects running on the same Agent that maintains the current CGI environment and other information about the current Web request or state-persistent WebSpeed transaction. The definitions of these variables reside in *install-path*`/src/web/method/cgidefs.i`.
- **WebSpeed preprocessor definitions**—A set of preprocessor names defined to provide consistent access to the Web environment, especially the Web output stream. The definitions of preprocessor names reside in *install-path*`/src/web/method/cgidefs.i`.
- **Function API**—A set of standard WebSpeed functions (user-defined SpeedScript functions) that provides a variety of services to Web objects. API functions handle low-level Web object tasks such as formatting URLs and returning specific values from the CGI environment. All API functions are available for your application. The source resides in several include (.i) files under *install-path*`/src/web/method`, including `cgiutils.i`, `cookies.i`, and `message.i`. Access to these functions in a Web object is provided by including *install-path*`/src/web/method/cgidefs.i`.
- **Method procedures**—A set of standard SpeedScript procedures that provides a variety of services to Web objects. Some method procedures that duplicate API functions are supported for upward compatibility with earlier WebSpeed versions. Others provide unique services to some types of Web objects, such as main-lining the response to Web requests. Some of these you can customize for each Web object. The source for many of the standard method procedures resides in the utility object `web-util.p` under *install-path*`/src/web/objects`.

- **Control handlers**—A special class of procedures that execute in response to pseudo events in HTML-mapping Web objects, and that you can customize. These pseudo event procedures include `web.input` and `web.output`. By overriding `web.input` and `web.output`, you can replace the default data movement between HTML field definitions and corresponding field objects for HTML-mapping Web objects. The default versions of these control handlers for each type of HTML field are located in the tagmap utilities under *`install-path`*`/src/web/support`. Each utility procedure understands how to convert data between a particular type of HTML form element and a SpeedScript field object of the corresponding type.

- **tagmap.dat**—A file that contains default mappings between HTML form element types and SpeedScript field object types for HTML-mapping Web objects. Each entry in the file includes the location of the default web.input and web.output control handler procedure for the field mapping. This file is also where you can define your own mappings and custom tags for your application. This file resides in your WebSpeed installation directory (*`install-path/`*).

- **web-disp.p**—The control program that runs on all WebSpeed Agents and executes all Web objects. The SpeedScript source resides in *`install-path`*`/src/web/object`. It manages various transaction states that can affect the whole application. This SpeedScript procedure is included with the development environment because it is central to the operation of WebSpeed applications. How `web-disp.p` manages Web objects can affect how and when you might set and evaluate transaction states in each Web object. This program also initializes the utility object `web-util.p` (residing in the same directory), where most API functions and method procedures reside at run time.

- **Sample database**—WebSpeed comes installed with Sports2000, a sample Progress database for trying out the example code in WebSpeed documentation.

2.5 WebSpeed Source Files

All SpeedScript source provided with WebSpeed, including `web-disp.p`, `web-util.p`, example Web objects, and other SpeedScript source files reside in directories under *`install-path`*`/src/web` and *`install-path`*`/src/web2`.

The source files in `install-path/src/web` are described below:

- **examples**—Contains the source for Web object examples described in this manual, including HTML and offset files. It also contains additional examples of interest.

- **method**—Contains source for various SpeedScript compile-time include files (similar in function to C include files) that define some of the method procedures and API functions used by WebSpeed to construct Web objects.

- **objects**—Contains source for the main Web object dispatch procedure `web-disp.p` and the utility object `web-util.p`.
- **support**—Contains all of the tagmap utility procedures for HTML-mapping Web objects that are specified in `tagmap.dat` and that define the default `web.input` and `web.output` control handlers for supported HTML form element and SpeedScript field object types. It also contains some run-time procedures for debugging and interpreting the offset file for an HTML-mapping Web object.
- **template**—Contains skeleton files used to create new Web objects and other types of support objects and files, including:
  - The file that defines the New File links that use these templates in the AppBuilder Files component (`web.cst`)
  - Blank embedded SpeedScript file template (`script.html`)
  - Embedded SpeedScript template for creating a database browser (`browse.html`)
  - Embedded SpeedScript template for creating an application welcome page (`main.html`)
  - HTML frame set template (`frameset.html`)
  - Embedded SpeedScript template for generating a table of database fields that can be executed by another Web object (`table.html`)
  - SpeedScript procedure file template (`procedur.p`)
  - SpeedScript include file template (`include.i`)
  - CGI Wrapper template for creating a SpeedScript-generating Web object (`wrap-cgi.w`)
  - HTML-mapping template and include files (`html-map.w` and `hmapmain.i`) for creating an HTML-mapping Web object

The source files in *install-path*/src/web2 are described below:

- **Super Procedures**—These are .i and .p files that contain the definitions of WebSpeed super procedures. To use super procedures you include the .i file in your 4GL code rather than running the .p file directly. Super procedures are in the *install-path*/src/web2 directory.

  For more information about super procedures, see the *Progress Programming Handbook*.

- **Templates**—Code relating to WebSpeed Report, Detail, and HTML Mapping Wizards is in the *install-path*/src/web2/templates directory. These files can be used as models for creating custom templates that can be made accessible to the AppBuilder through a .cst file.

  There are two kinds of files in the template directory. First are the Progress template files (.w and .i files). These are used to create new objects that can be made available from the AppBuilder palette. More information on using Progress template files to extend AppBuilder can be found in the *Progress AppBuilder Developer's Guide*.

  The second type of file is the HTML template files (.dat files) for the report and detail wizards. These can be used as examples of how to create new wizards that create Embedded 4GL files (.html files). Each .dat file has a companion .w file with the same file name (for example, wreport.w and wreport.dat) that acts as a SpeedScript wrapper for the .dat file.

3

WebSpeed Architecture

This chapter is an overview of the WebSpeed architecture, which includes the following:

- Definition of WebSpeed components and how they can be distributed in a network environment.
- How WebSpeed handles a request
- Configuration and management tools

3.1 WebSpeed Components

The WebSpeed components include:

- **WebSpeed Agent**—A process that executes the Web objects, performs database transactions, and dynamically merges data into HTML format. This is the standard character Progress 4GL client running in batch mode.
- **WebSpeed Broker**—A process that does the following:
  - Registers the WebSpeed Services that it provides to a NameServer for access by one or more HTML clients. The HTML client runs from within an Internet browser.
  - Manages connections between clients and a pool of WebSpeed Agents.
  - Maintains the status of each agent in its pool and dynamically scales the number of agents according to changing demand.
- **WebSpeed Messenger**—A process that handles the transfer of data between the Web server and the WebSpeed Agent during a single Web transaction. The Messenger is a CGI program or an ISAPI or NSAPI process, depending on the Web server and how you plan to write your applications.
- **NameServer**—A process that maintains a list of WebSpeed Transaction Servers. The Transaction Servers register the application services they provide with the NameServer. The NameServer can then direct client connection requests to a WebSpeed broker that supports a requested application service. This provides scalability and location transparency to your applications.

  When the Enterprise Transaction Server is installed, the NameServer also performs load balancing. Load balancing allows you to balance client workload among multiple brokers that support the same application service (that is, the same set of procedures and resources).
- **WebSpeed Workshop Environment**—The tools that you use to develop and test WebSpeed applications. See Chapter 2, "Overview of WebSpeed Tools" for more information.

3.1.1 WebSpeed Component Distribution

Table 3–1 shows the maximum component distribution that you can achieve with WebSpeed components over a network. It shows the major components of the WebSpeed run-time model that must reside together on the same machine. The WebSpeed Messenger must live on the same machine as the Web server. The WebSpeed Broker, and Agent, also reside together on the same machine. Otherwise, you can combine these configurations together in any combination of one or more machines that WebSpeed supports.

Table 3–1: Maximum WebSpeed Component Distribution

Machine 1	Machine 2	Machine 3	Machine 4	Machine 5
WebSpeed Workshop	Web Server WebSpeed Messenger	NameServer	WebSpeed Broker WebSpeed Agent	DB Server

In addition to these, you can also add a Progress AppServer to any supported machines on the network. An AppServer can execute Progress procedures on behalf of a WebSpeed Agent acting as a Progress client. For more information on the Progress AppServer, see the *Building Distributed Applications Using the Progress AppServer* manual.

For more detailed information about WebSpeed component distribution, see the *WebSpeed Installation and Configuration Guide*.

3.1.2 WebSpeed Transaction Server Products

You may see the WebSpeed Transaction Server referred to as the WebSpeed Development Server, the WebSpeed Transaction Server, or the WebSpeed Enterprise Transaction Server. All of these WebSpeed products are comprised of the same components. However, there are differences in the number of WebSpeed Agents each supports. In addition, only the WebSpeed Enterprise Transaction Server supports load balancing. Table 3–2 summarizes the differences.

Table 3–2: WebSpeed Products

Product Name	Number of Agents Supported	Load Balancing
WebSpeed Development Server	2	No
WebSpeed Transaction Server	5 or 25	No
WebSpeed Enterprise Transaction Server	50 or 250	Yes

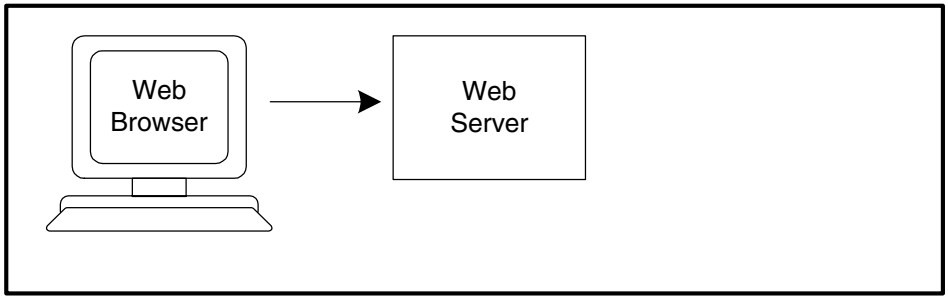
Since all three WebSpeed products contain essentially the same components, you can use them all to develop and deploy WebSpeed applications. However, you will need more WebSpeed Agents in a deployment environment than in a development environment. The WebSpeed Agent runs Web objects in response to client requests, and there will probably be fewer client requests in a development environment.

The WebSpeed Enterprise Transaction Server supports load balancing, which increases its efficiency in handling client requests. Load balancing is implemented by allowing the NameServer to manage multiple WebSpeed Transaction Servers, which can be distributed on multiple machines in a network environment.

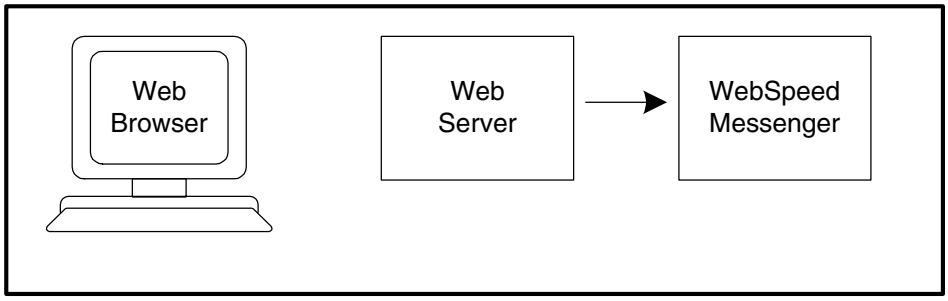
3.2 How WebSpeed Handles a Request

This section describes how the components of the WebSpeed Transaction Server interact to respond to a request from a client browser. The basic request cycle is the same for the WebSpeed Development Server, Transaction Server, or the Enterprise Transaction Server. The basic request cycle is also the same for all types of Web objects.

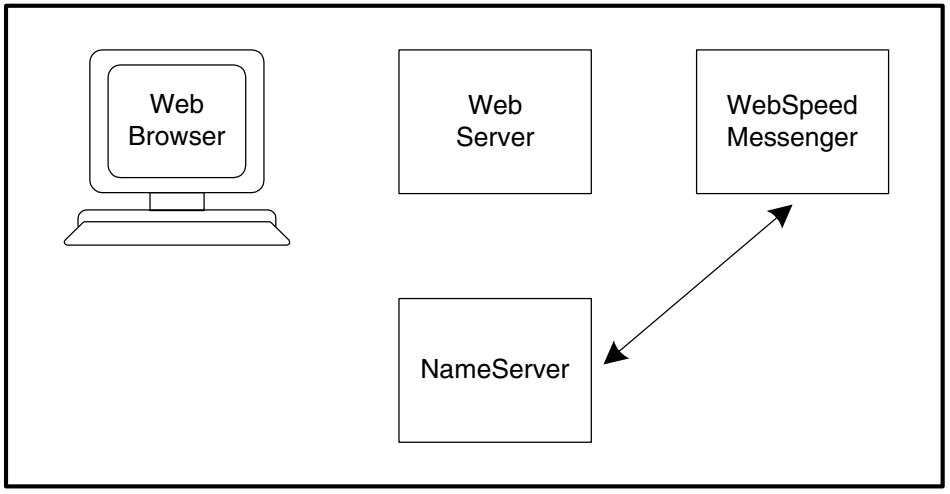
1. The Web server receives the request from an Internet browser. Each request is in the form of a URL:



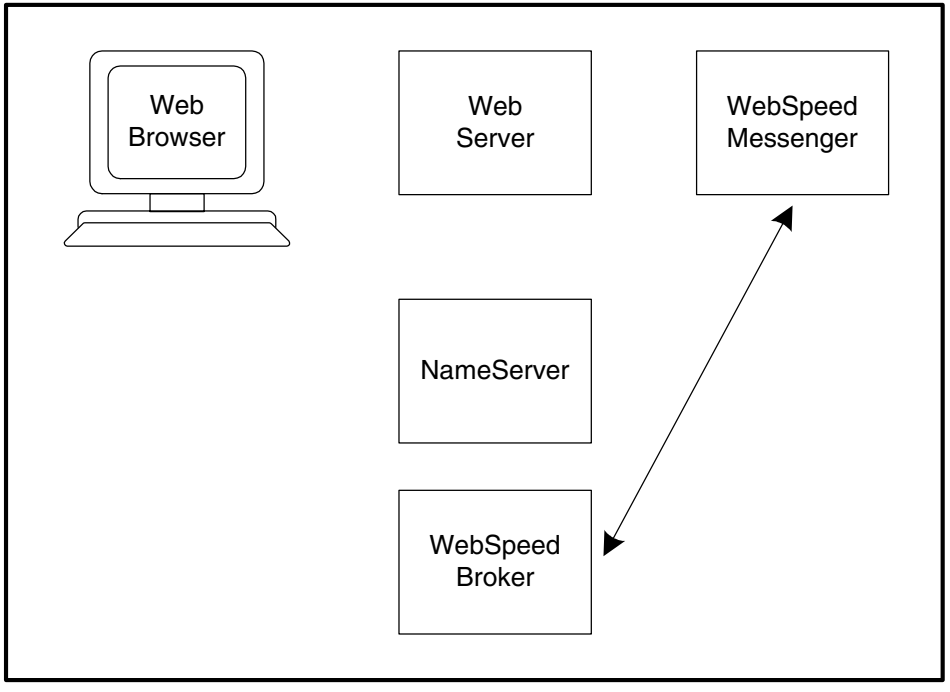
2. The Web server spawns a CGI Messenger or wakes up a previously spawned ISAPI or NSAPI Messenger:



3. The WebSpeed Messenger sends a request to the NameServer for an available WebSpeed Broker:

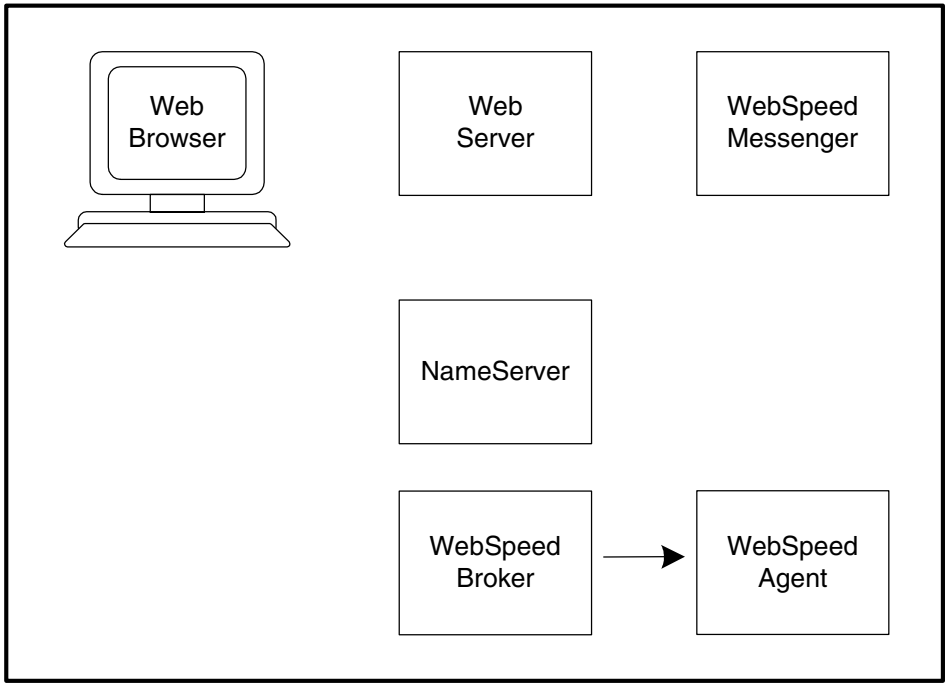


4. The Messenger initiates a connection to the WebSpeed Broker at the address returned by the NameServer and forwards the request to that Broker:

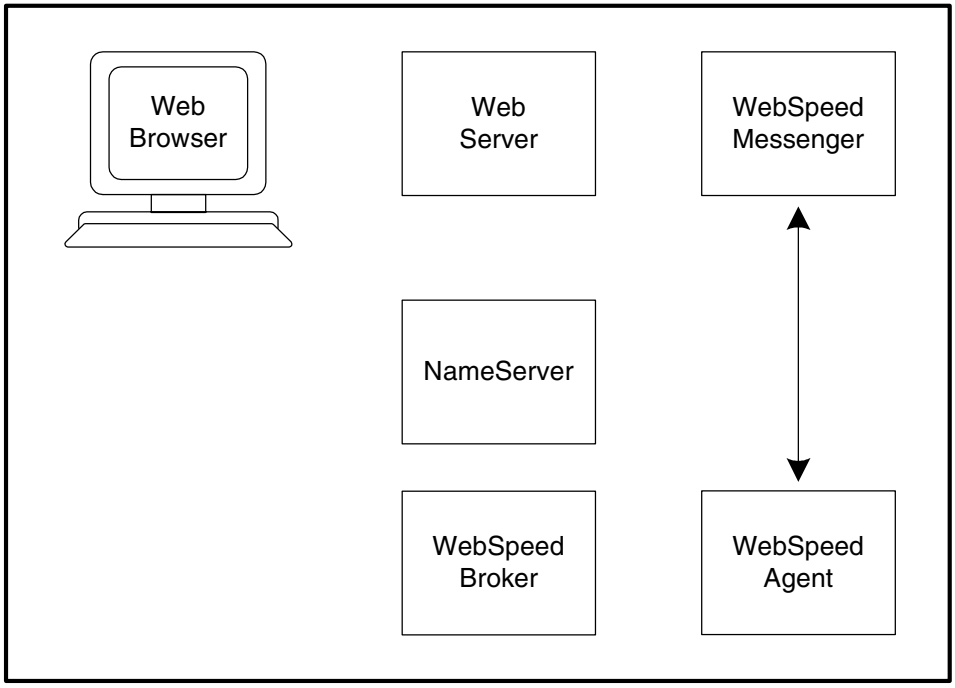


NOTE: Under the WebSpeed Enterprise Transaction Server, the NameServer can manage multiple WebSpeed Brokers that support the same application service. It will choose a broker based on load-balancing criteria.

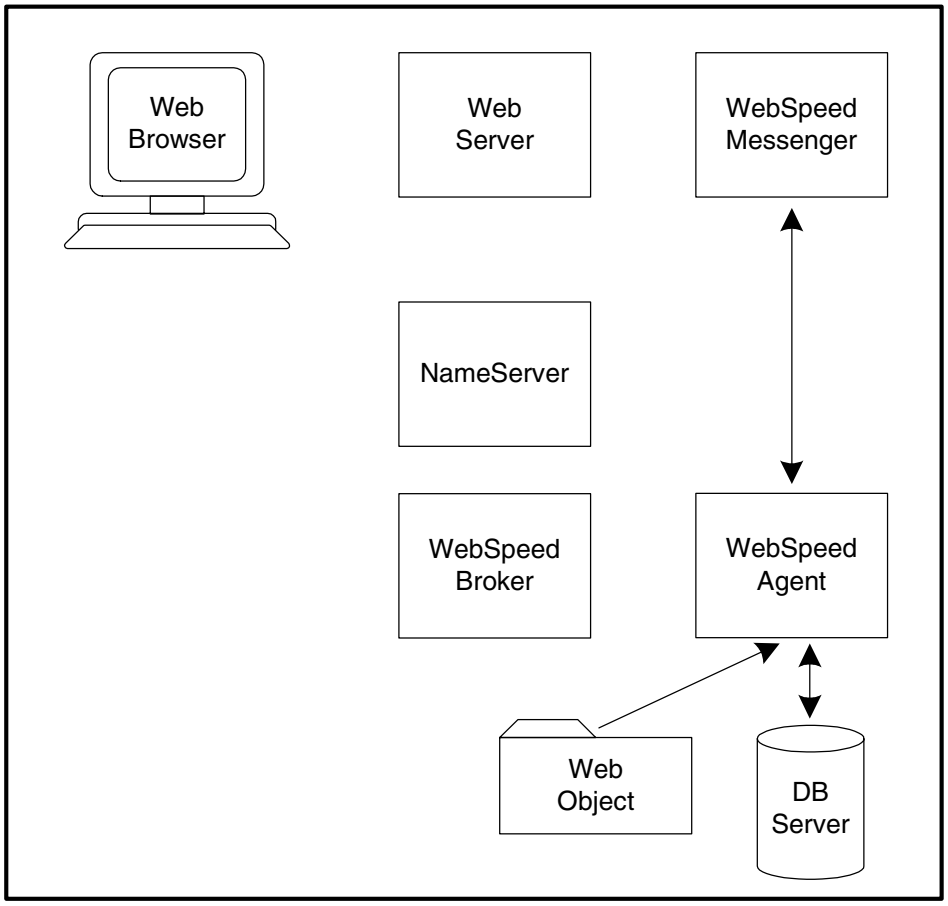
5. The WebSpeed Broker consults its pool of WebSpeed Agents and assigns the request to an idle Agent:



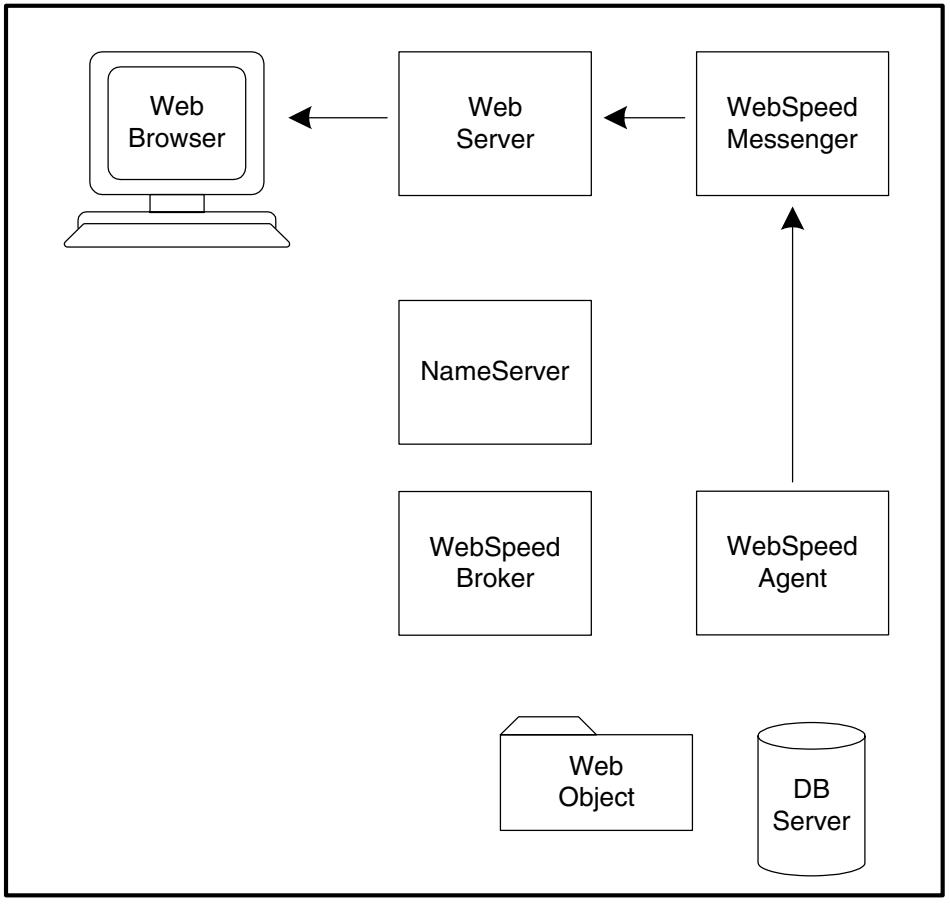
6. The WebSpeed Agent and the WebSpeed Messenger establish a connection. The Web object name and the environment for the request are passed to the Agent:



7. The WebSpeed Agent executes the Web object. It also performs the necessary reads from and writes to the database. Notice that the WebSpeed Agent maintains its connection with the WebSpeed Messenger while the Web object is executing:



8. The WebSpeed Agent passes the generated HTML Web Page to the Messenger, which sends it to the browser through the Web Server:



A state-aware Web object, although no longer executing, maintains a persistent context for later execution by the Agent. For more information about stateless and state-aware Web objects, see Chapter 8, "Controlling WebSpeed Transactions."

After completion, the Agent disconnects from the Messenger, and the Agent updates its availability with the Broker. The Web server disconnects from the browser after sending the HTML Web Page.

3.3 WebSpeed Configuration and Management

This section provides an overview of the management tools for WebSpeed. For more detailed information, see the *WebSpeed Installation and Configuration Guide*.

3.3.1 Progress Explorer

The Progress Explorer is a graphical tool that combines the functionality of all the command-line utilities. You can use it to start, create, delete, and modify WebSpeed Brokers NameServers, and Messengers. You can also use Progress Explorer to start, create, delete, and modify database servers.

Currently, Progress Explorer runs only on Windows operating systems. However, you can connect to remote UNIX systems and administer the WebSpeed components running on those remote UNIX systems.

See the *WebSpeed Installation and Configuration Guide* or the Progress Explorer online help for more information.

3.3.2 Command-line Utilities

Command-line utilities are available for both the UNIX and Windows operating systems.

The WTBMAN utility allows you to control the operation and query the status of a configured WebSpeed Transaction Server. The WSCONFIG utility displays and validates property settings for the WebSpeed Transaction Server and the WebSpeed Messenger

The NSMAN utility allows you to control the operation and query the status of a configured NameServer. The NSCONFIG utility displays and validates property settings for the NameServer.

The PROADSV utility (on UNIX systems only) allows you to start up and shut down the AdminServer.

For more information about these command-line utilities, see the *WebSpeed Installation and Configuration Guide*.

3.3.3 SpeedStart

SpeedStart is a browser-based reference page that contains a list of URLs that you can use to check if WebSpeed components are running properly. You can access SpeedStart from your Windows Start Menu (Start→ Programs→ Progress→ SpeedStart).

SpeedStart also contains a general list of WebSpeed requirements, plus links to installation and configuration information.

4

Sample Web Objects

This chapter describes several sample Web objects that are included with WebSpeed. These examples represent the basic types of Web object that you can build with WebSpeed. In addition, the examples illustrate some basic WebSpeed techniques and concepts. Topics include:

- An overview of the types of Web object
- Instructions on how to use Web object examples
- Embedded SpeedScript examples
- A CGI Wrapper example
- HTML Mapping examples

4.1 Types of Web Objects

A WebSpeed Web object is an external procedure (a .r file) that can be invoked by a URL. Web objects generate and update web pages, and may also interact with data sources. WebSpeed applications are composed of one or more Web objects.

WebSpeed Web objects may be categorized according to the content of their sources. The four basic categories of Web objects are:

- Standard HTML
- HTML with embedded SpeedScript
- CGI Wrapper
- HTML Mapped

All four types of source files, when compiled, become executable r-code that can be run by a WebSpeed Agent.

4.1.1 Standard or Static HTML Web Objects

Using WebSpeed, you can compile almost any HTML file into a Web object. The compilation process creates a temporary .w file and then produces the executable .r file. The resulting Web object (that is, the r-code) generates a Web page that is identical to the original HTML file when viewed in a browser.

At first, it might seem unnecessary to convert HTML files into Web objects since HTML files are already viewable in a browser. However, when you run a WebSpeed application, you are running Web objects on a WebSpeed Agent. WebSpeed Agents cannot process HTML files directly; they can only execute r-code. Therefore, simple or static HTML files must be first converted to r-code so they can be re-generated by a WebSpeed Agent.

Your WebSpeed application is likely to be composed of some relatively simple Web objects, (derived from static HTML files) combined with more dynamic and interactive Web objects.

4.1.2 Embedded SpeedScript Web Objects

SpeedScript is a subset of the Progress 4GL language. Embedded SpeedScript Web objects are compiled from HTML files that contain SpeedScript code. The SpeedScript code is embedded between the `<script Language="SpeedScript">` tag and the `</script>` end tag in an HTML file.

Compilation of embedded SpeedScript files is similar to the compilation of standard HTML files. The compilation process creates a temporary .w file and then produces an executable .r file. The r-code recreates the content of the original HTML file and adds the dynamic content or any additional processing logic created by the embedded SpeedScript code.

The Detail Wizard and the Report Wizard in AppBuilder both create r-code using embedded SpeedScript in an HTML file. In addition, the AppBuilder contains a Blank Template which is an HTML file with markup for embedded SpeedScript.

See the "Embedded SpeedScript Examples" section for some examples. Also see the "Overview of Embedded SpeedScript" section in Chapter 5, "SpeedScript" for more information about embedded SpeedScript.

4.1.3 CGI Wrapper Web Objects

The source for CGI Wrapper Web objects is a SpeedScript .w file. There is no HTML source file associated with a CGI Wrapper Web object. Compilation of the .w source file produces a .r file that is executed by a WebSpeed Agent. When executed, CGI Wrapper Web objects dynamically create HTML content that is returned to the client browser.

The source for CGI Wrapper Web objects contains HTML markup that is "wrapped" by a SpeedScript {&OUT} preprocessor statement. The {&OUT} preprocessor combines with a subprocedure called `process-web-request` to generate a valid HTML page. The generated page includes an HTML header that is produced by the CGI Wrapper.

The AppBuilder includes a CGI Wrapper template. This template provides a skeleton SpeedScript file that contains the basic SpeedScript code to generate and output an HTML file to the Web. You start by coding directly in SpeedScript and generate all of the HTML from within the executable SpeedScript file.

See the "CGI Wrapper Example" section for more information.

4.1.4 HTML Mapped Web Objects

Creation of a HTML Mapped Web object begins with a standard HTML file (.htm or .html) that contains form elements. A SpeedScript procedure file (.w) maps the form elements to a database field or to a SmartDataObject data element. Compilation produces an offset file (.off) before producing the executable r-code file (.r). The offset file records the locations of form fields based on the layout of the HTML source file.

Complex data manipulation can be performed using an HTML Mapped Web object. Much of the logic needed to perform database updates is included as the default behavior of an HTML Mapped Web object.

The AppBuilder HTML Mapping wizard provides a quick way to map HTML form fields to database fields or SmartDataObject data elements.

See the "HTML Mapping Examples" section for more information.

4.1.5 General Criteria for Use

The different types of Web objects provide different approaches to Web development, but they also provide different WebSpeed capabilities. Your choice of Web object depends both on your development approach and on what you want your application to do.

If you feel most comfortable working directly in HTML, you probably want to use embedded SpeedScript Web objects.

If you want to separate user interface from internal logic or gain more programmatic control over internal logic, you might want to use CGI Wrapper or HTML Mapped Web objects. To build state-persistent applications, you must use CGI Wrapper or HTML Mapped Web objects. See Chapter 8, "Controlling WebSpeed Transactions," for more information about state-persistence.

Note also that each type of Web object uses different mechanisms for merging HTML and program logic. Embedded SpeedScript and CGI Wrapper Web objects merge the HTML with program logic at compile time. HTML Mapped Web objects merge the HTML almost completely at run time. With compile-time merging, you must code all HTML input and output. Compile-time merging also offers higher run-time performance, but incurs limitations on effective HTML file size. All of the compiled HTML is added to the single text segment that includes SpeedScript character expressions (strings), and this segment is limited in size (60KB). Alternately, the run-time merging of HTML-mapping provides a cleaner separation between HTML authoring and SpeedScript programming, and much of the HTML input and output is automated for simple merges. However, to generate Web pages with complex dynamic interactions, the programming can be much more complex than in an equivalent embedded SpeedScript Web object.

The following sections describe examples of the basic Web object types. They also illustrate some of the basic features of the SpeedScript that you use to build all Web objects.

4.2 Using Web Object Examples

To view or run the example Web objects mentioned in this chapter, do the following:

1 ♦ Make a working copy of *install-path*/src/web/examples.

If you make a working copy of the examples, you can restore them if you accidentally overwrite or delete them.

2 ♦ Start WebSpeed AppBuilder.

3 ♦ Choose Tools→ Database Connections and connect to a working copy of the Sports2000 database.

The Sports2000 database is a sample Progress database installed in your installation directory. To run the Web object examples described in this chapter, you should make a working copy of Sports2000 and then start a database server for it. See *Getting Started with WebSpeed* for detailed instructions on copying and starting the Sports2000 database.

4 ♦ Choose Options→ Preferences, click on the WebSpeed tab, and specify your Web browser and your Broker URL.

5 ♦ Choose Tools→ WebTools and then select File Tools from the left frame of the browser window.

6 ♦ From the File Tools interface, navigate to your working copy of the `examples` directory that you created in Step 1.

You will see the example Web objects in a file list. You can view, compile, or run them from the File Tools interface.

If you want to edit the examples, open them by selecting File→ Open from the AppBuilder main menu. Then navigate to the directory that contains your working copy of the examples. Be sure to edit a working copy of an example and not the original.

Templates for the various Web object types are available when you choose File→ New from the AppBuilder main menu.

In addition, you can view sample WebSpeed applications. Each sample application is composed of several Web objects. The sample applications are available from WebTools help. From the AppBuilder, choose Tools→ WebTools and then select Help. See *Getting Started with WebSpeed* for more information about WebSpeed sample applications.

4.2.1 Web Object URLs and PROPATH

You use standard URL notation to access Web objects from a browser. The following shows the general syntax:

```
http://host_name [:port] /scripts_dir/messenger/WService=broker/web_object
```

host_name

Specifies the Web server host name.

port

Specifies the port number of the Web Server. The port number is optional if the Web Server uses the default port number, which is 80. For example, if a Web Server's port number is 88, the initial part of the URL might be specified as/ `http://myhost:88`.

scripts_dir

Specifies the Web server scripts directory for a CGI or ISAPI Messenger. Omit this component if you using an NSAPI messenger.

messenger

Specifies the messenger name. For a CGI Messenger, you add the filename (or pathname relative to the scripts directory) of your CGI Messenger script. For an ISAPI Messenger, you add the filename (or pathname relative to the scripts directory) of the DLL (`wsisa.dll`). For an NSAPI messenger, you add the filename of the DLL (`wsnsa.dll`).

broker

Specifies the WebSpeed broker name.

web_object

Specifies the Web object name.

Finally, you add the pathname of the requested Web object, relative to the WebSpeed PROPATH environment variable setting for your WebSpeed Agents. The PROPATH setting is a list of directories or procedure libraries that a WebSpeed Agent searches left-to-right in order to execute a Web object (or other SpeedScript procedure file). The Messenger passes the Web object pathname to the Agent as part of the CGI environment (in PATH_INFO). The Agent control program (`web-disp.p`) then searches for and executes the specified Web object or associated r-code.

For more information on PROPATH and WebSpeed URLs, see your *WebSpeed Installation and Configuration Guide*.

4.3 Embedded SpeedScript Examples

You can author embedded SpeedScript files using most HTML authoring tools, then compile them in the AppBuilder. In addition, you can build them entirely in AppBuilder. The Report and Detail wizards provide an automated way of creating embedded SpeedScript objects that include logic that references a particular data source.

To execute the embedded SpeedScript examples in this section, you should first copy them to your working directory and compile them in the AppBuilder. For more information on using examples see the "Using Web Object Examples" section.

4.3.1 A Simple Query

The following shows the content of `w-sstcst.html`, which is one of the Web object example files found in the example directory:

w-sstcst.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<title>Sports Customer List</title>
</head>

<body BGCOLOR="#FFFFFF">
<h1>Customer List</h1>
<table border>
    <tr>
        <th>Customer ID</th>
        <th>Customer Name</th>
        <th>Phone Number</th>
    </tr>
    <script language="SpeedScript">
        FOR EACH Customer:
    </script>
    <tr>
        <td align=left>'CustNum'</td>
        <td align=left>'Name'</td>
        <td align=left>'Phone'</td>
    </tr>
    <script language="SpeedScript">
        END.
    </script>
</table>
</body>
</html>
```

This example illustrates the two basic WebSpeed tags used with embedded SpeedScript:

- Statement escape tags (`<script language="SpeedScript"> ... </script>`), which allow you to embed one or more complete SpeedScript statements.

  In this example the embedded SpeedScript includes two SpeedScript statements, FOR EACH and END. These two statements form an iterating block that reads each record in the Customer table. For each record read, the block also executes the HTML markup between the FOR EACH statement and the END statement.

- Expression escape tags (‘...‘), which allow you to include the current value of a single data element directly into your HTML source.

  NOTE: The expression escape tags (‘...‘) are back-tick characters and are not single quotes.

  In this case, the SpeedScript writes the values of three fields from the Customer table (CustNum, Name, and Phone) as character strings. The HTML markup formats the character strings into a table, one row per record.

Figure 4–1 shows the output when you compile and run `w-sstcst.html`.

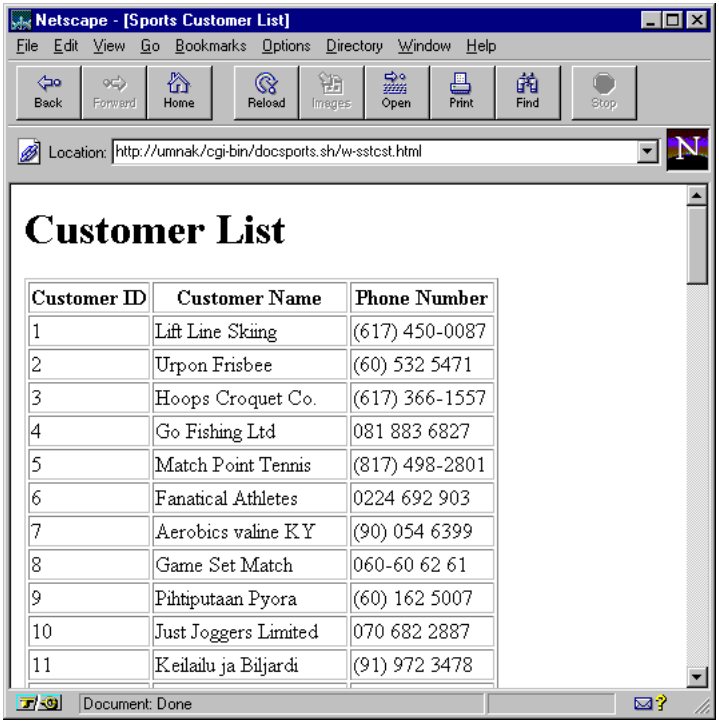


Figure 4–1: Running w-sstcst

WebSpeed supports several alternative tags for both statement and expression escapes to ensure that you can use tags acceptable to your authoring tool. For more information, see the “Statement Escapes” and “Expression Escapes” sections in Chapter 5, “SpeedScript.”

4.3.2 Handling Form Input

The following shows the section of `w-sstget.html`, which uses embedded SpeedScript to handle input from an HTML form:

w-sstget.html

```
.
.
.
<H1>Customer Information</H1>
  <!--WSS
  IF get-value("CustNum") <> "" THEN
    FIND Customer WHERE CustNum = INTEGER(get-value("CustNum"))
      NO-ERROR. -->
  <FORM ACTION="w-sstget.html" METHOD="POST">
    <P>
    Enter Customer ID: <INPUT NAME="CustNum" SIZE="5"
      VALUE='IF AVAILABLE(Customer) THEN STRING(CustNum) ELSE ""'>
      <INPUT TYPE="SUBMIT" NAME="Button" VALUE="Find">
    <BR><BR>
    <!--WSS
    IF AVAILABLE(Customer) THEN DO: -->
      Name: <INPUT NAME="Name" SIZE="20" VALUE="'Name'">
      --------Phone: <INPUT NAME="Phone" SIZE="20" VALUE="'Phone'">
       <BR><BR>
      Address 1: <INPUT NAME="Address" SIZE="20" VALUE="'Address'"><BR>
      Address 2: <INPUT NAME="Address2" SIZE="20" VALUE="'Address2'">
      <BR><BR>
      City: <INPUT NAME="City" SIZE="12" VALUE="'City'">
      ---State: <INPUT NAME="State" SIZE="20" VALUE="'State'">
      ---Zip: <INPUT NAME="PostalCode" SIZE="10" VALUE="'PostalCode'">
      <BR><BR>
      Comments: <TEXTAREA NAME="Comments" ROWS="2" COLS="30">
        'Comments'</TEXTAREA>
    <!--WSS
    END. -->
    </P>
  </FORM>
</BODY>
</HTML>
```

This HTML file uses a different set of statement escape tags (`<!--WSS ... -->`) than the previous example, but they have the same effect.

The first IF...THEN statement invokes the FIND statement to read a customer record if a customer number (CustNum) is entered by the user. This user enters a customer number in the Web page initially generated by the Web object, as shown in Figure 4–2.

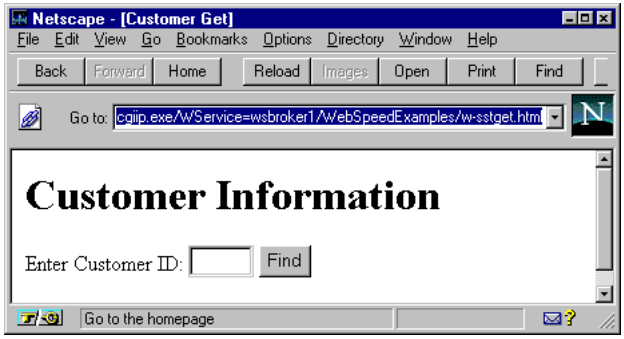


Figure 4–2: User Input Page Generated by w-sstget

This input is tested by the get-value function, which is one of the WebSpeed API functions that facilitate IO between Web pages and Web objects. The get-value function returns the value of a named element passed with a request, including a form element, a query string element, a cookie, or a user field (an internally defined list value). In this case, the function looks for the value of the CustNum form field defined in the HTML.

For more information on WebSpeed API functions, see the AppBuilder online help. Choose Help→ Help Topics from the AppBuilder menu bar. Then select the Find tab and enter WebSpeed API in the top field of the dialog box. Also see "WebSpeed API Functions" in Chapter 5, "SpeedScript," for an overview of some commonly used API functions.

NOTE: The NO-ERROR option on the FIND statement allows the Web object to continue execution if the FIND statement fails because the CustNum value is invalid.

The HTML for the CustNum form element (field) also includes an expression escape tag to provide the field value. The expression contains the SpeedScript IF...THEN...ELSE function, which returns a value based on a condition. Here, the function tests whether the Web object found a customer record by returning the value of the SpeedScript AVAILABLE function on the Customer table (TRUE or FALSE). The AVAILABLE function value is always FALSE for the initial request and a null string value is returned for the IF...THEN...ELSE function.

The second IF...THEN statement also checks AVAILABLE(Customer), and generates HTML to display several Customer field values only if there is a customer record available. This second IF...THEN statement encloses the relevant HTML in a DO block terminated by an END statement to ensure that all the HTML is generated for the available customer record.

Figure 4–3 shows the HTML output as it would appear in a browser.

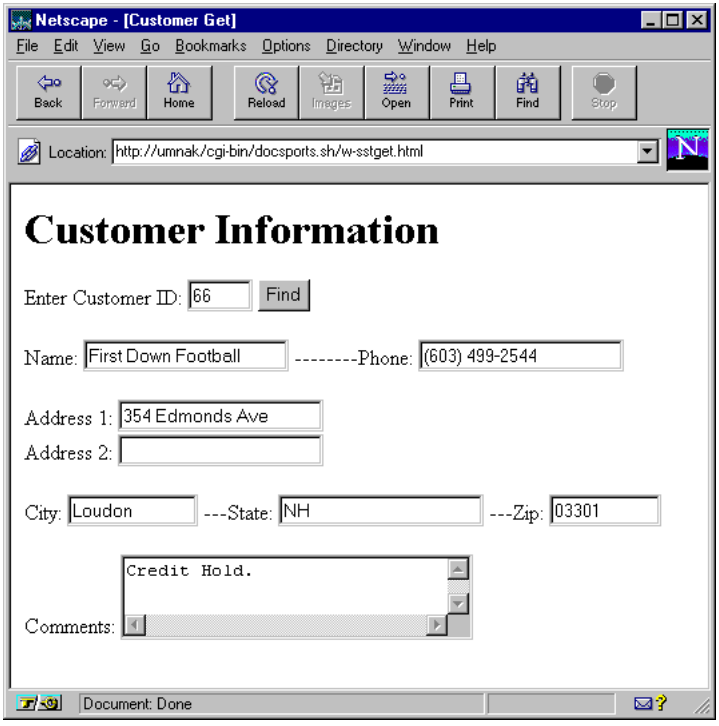


Figure 4–3: Customer Information Generated by w-sstget

NOTE: Although expression escape tags output all SpeedScript expressions as character strings, the values that combine to make a SpeedScript expression can have other data types, such as integer. SpeedScript requires compatible data types in expressions. In the example, the SpeedScript INTEGER function converts the character value of the get-value API function to an integer in order to allow comparison with the integer CustNum field in the Customer table. Similarly, the SpeedScript STRING function converts the integer value of CustNum to a character string in the IF...THEN...ELSE function, because the alternative values for the function must have compatible data types.

As these examples show, you can achieve a fine degree of control over HTML generation within an embedded SpeedScript Web object.

4.4 CGI Wrapper Example

The following shows an extract of the process-web-request procedure in the example CGI Wrapper Web object, `w-forcst.w`.:

w-forcst.w

```
PROCEDURE process-web-request :

  RUN output-header.

  {&OUT}
    "<HTML>":U SKIP
    "<HEAD>":U SKIP
    "<TITLE>Sports Customer List</TITLE>":U SKIP
    "</HEAD>":U SKIP
    "<BODY BGCOLOR=~"#FFFFFF~">":U SKIP
    .

  /* Output your custom HTML to WEBSTREAM here (using {&OUT}).      */

  {&OUT}
    "<H1>Customer List</H1>":U SKIP
    "<TABLE BORDER>":U SKIP
    "<TR>":U SKIP
    "   <TH>Customer ID</TH>":U SKIP
    "   <TH>Customer Name</TH>":U SKIP
    "   <TH>Phone Number</TH>":U SKIP
    "</TR>":U SKIP
    .

  FOR EACH Customer:
    {&OUT} "<TR>":U SKIP
           "    <TD ALIGN=LEFT>":U CustNum "</TD>":U SKIP
           "    <TD ALIGN=LEFT>":U Name "</TD>":U SKIP
           "    <TD ALIGN=LEFT>":U Phone "</TD>":U SKIP
           "</TR>"
           .
  END.

  {&OUT}
    "</TABLE>":U SKIP
    "</BODY>":U SKIP
    "</HTML>":U SKIP
    .

END PROCEDURE.
```

This Web object is equivalent to `w-sstcst.html`, but instead of HTML it is written entirely in SpeedScript. The example shows a single procedure from `w-forcst.w` (omitting some comments) called process-web-request. (A *procedure* is a block of SpeedScript code that you execute by name using a RUN statement. Its capabilities are like those of procedures in Pascal or of subroutines in other languages.)

The process-web-request procedure is executed in every SpeedScript Web object that responds to a Web request and contains the main-line code for satisfying the request. Thus, process-web-request also functions as a method of the Web object. WebSpeed provides two different templates for process-web-request, one for CGI Wrapper Web objects and one for HTML-mapping Web objects.

NOTE: A *method* performs a generic action associated with an object, and is available to perform that action for all instances of the object. In WebSpeed, many procedures function like methods and are therefore known as *method procedures*.

Aside from being coded entirely in SpeedScript, the main functional difference between `w-forcst.w`. and `w-sstcst.html` is the process-web-request method procedure. This procedure allows a Web object to make itself state aware and thus manage state-persistent WebSpeed transactions. For more information on WebSpeed transactions and Web object states, see Chapter 8, "Controlling WebSpeed Transactions."

In this case, process-web-request generates the entire Web page to satisfy the request, including all HTML and associated data. First, it runs the output-header procedure to generate the HTTP header for the page. It then reads the Customer table of the sample Progress Sports2000 database (using the FOR EACH statement). It builds an HTML table listing the ID number, name, and phone number for each Customer record in the database table.

The end result is a tabulated list displayed as a page on the Web browser. It is identical to the output from `w-sstcst.html` that is shown in Figure 4–1.

To create a CGI Wrapper Web object in the AppBuilder, you select CGI Wrapper in the File→ New dialog box. A template comes up in the AppBuilder Section Editor. The code in bold text in `w-forcst.w`. shows all the code that you enter to create it. The code between quotation marks is plain HTML entered as SpeedScript character strings.

The {&OUT} symbol is WebSpeed syntax that initiates output to a Web page using a preprocessor variable. You can find the actual SpeedScript definition for OUT (the preprocessor name) in the *install-path*`/src/web/method/cgidefs.i` file installed with WebSpeed. This is an include file (source code file) shared by all Web objects. For more information about {&OUT}, see the *Progress Programming Handbook*.

A SpeedScript statement is always terminated by a period (.) or colon (:); so a period terminates each chunk of HTML started with the {&OUT} reference. The :U is a flag that tells the Progress translation tool, Translation Manager, not to select this particular string for translation. The SKIP option begins a new line in the Web page. Notice the use of the tilde (~), the SpeedScript escape character, in the BGCOLOR attribute statement. The interior quotation marks are preceded by the escape character so they will not be misinterpreted as end quotes.

4.5 HTML Mapping Examples

HTML mapping allows you to take an existing HTML form layout and map (or link) specific form fields to specific database fields. The generated Web object, a SpeedScript procedure file, provides processing to manage the input and output of values between the mapped source and the target fields.

With HTML mapping, you can separate the design of your HTML interface from the development of the SpeedScript procedures that interact with your data source. Also, like CGI Wrapper Web objects, HTML Mapping Web objects allow you to initiate and control state-persistent WebSpeed transactions. See Chapter 8, "Controlling WebSpeed Transactions," for more information about Web object states.

However, the programming and maintenance of HTML-mapping Web objects is usually more complex than Web objects derived from embedded SpeedScript.

The following examples show relatively simple and more complex HTML-mapping Web objects.

4.5.1 Simple HTML Mapping

HTML mapping usually begins with a file containing HTML form markup, like the example `w-csget.htm`:

w-cstget.htm

```
<!DOCTYPE HTML PUBLIC "-//Netscape Corp.//DTD HTML plus
Tables//EN" "html-net.dtd">

<HTML>
<HEAD>
<TITLE>Customer Get</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Customer Information</H1>
<FORM ACTION="w-cstget.w" METHOD="POST">
<P>
Enter Customer ID: <INPUT NAME="CustNum" SIZE="5">
<INPUT TYPE="SUBMIT" NAME="Button" VALUE="Find"><BR><BR>
Name: <INPUT NAME="Name" SIZE="20">
--------Phone: <INPUT NAME="Phone" SIZE="20"><BR><BR>
Address 1: <INPUT NAME="Address" SIZE="20"><BR>
Address 2: <INPUT NAME="Address2" SIZE="20"><BR><BR>
City: <INPUT NAME="City" SIZE="12">
---State: <INPUT NAME="State" SIZE="20">
---Zip: <INPUT NAME="PostalCode" SIZE="10"><BR><BR>
Comments: <TEXTAREA NAME="Comments" ROWS="2" COLS="30"></TEXTAREA>
</P>
</FORM>
</BODY>
</HTML>
```

Note that this file is equivalent to `w-sstget.html`. However, it does not contain any embedded SpeedScript. (Also notice that the call to the Web object `w-cstget.w` is an attribute of the FORM element in `w-csget.htm`.) Instead of embedding SpeedScript in this file, you can use the HTML Mapping Wizard in the AppBuilder to:

- Map each INPUT and TEXTAREA form element to a field in the Sports2000 database.
- Generate an offset file that specifies the position of the form elements in the HTML that is returned to the client.
- Edit process-web-request, which controls the flow of your Web object logic, by adding a SpeedScript FIND statement.
- Create `w-cstget.w` by compiling and saving in the AppBuilder.

Mapping Form Elements to Database Fields

You begin by choosing File→ New from the AppBuilder main menu, and then selecting HTML Mapping from the list of Web objects. The wizard guides you through the process of specifying an HTML file and a database. (When you specify the Sports2000 database for this example, you must add the Customer table in the wizard's Query Builder.) Then, it allows you to map the form elements to database fields. The wizard also has an Automap feature that can map form elements to database fields that have similar names.

Generating the Offset File

When the HTML Mapping Wizard completes, it automatically creates an offset file that contains the type and location of each form element in the HTML file. The offset file generated from `w-csget.htm` is called `w-cstget.off` and looks similar to the following:

w-cstget.off

```
/* HTML offsets */
htm-file= C:\PROGRESS\WRK\examples\w-cstget.htm
version=

field[1]= "CustNum,INPUT,,fill-in,11,20,11,50"
field[2]= "Name,INPUT,,fill-in,13,7,13,35"
field[3]= "Phone,INPUT,,fill-in,14,16,14,45"
field[4]= "Address,INPUT,,fill-in,15,12,15,43"
field[5]= "Address2,INPUT,,fill-in,16,12,16,44"
field[6]= "City,INPUT,,fill-in,17,7,17,35"
field[7]= "State,INPUT,,fill-in,18,11,18,40"
field[8]= "PostalCode,INPUT,,fill-in,19,9,19,43"
field[9]= "Comments,TEXTAREA,,editor,20,11,20,66"
```

When it runs, an HTML mapped Web object uses the offset file to determine where to insert data values in the HTML that it generates. The numbers at the end of each field line in `w-cstget.off` indicate where the form field begins and ends in the `w-csget.htm` source file.

NOTE: If you make **any** changes in your HTML source files, you will probably need to generate a new offset file. If you add a single carriage return to your HTML source, for example, all of positional information in an existing offset file might be invalid. Although you could update the offset file yourself, manual editing is error prone. It is safer to use the HTML Mapping Wizard and to generate a new offset file.

You can automatically re-map by opening the Web object in the AppBuilder. An updated offset file will be generated. The WebSpeed Transaction Server also automatically generates the offset file at run time if the file does not exist or is out of date.

When the HTML Mapping Wizard completes and saves the offset file, you will see the tree view representation of the object, as shown in Figure 4–4.

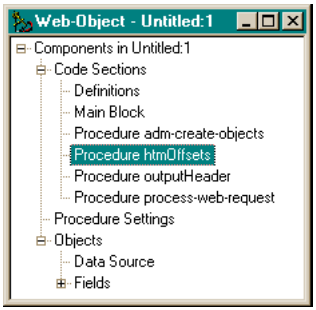


Figure 4–4: Tree View of HTML Mapping Source

By selecting a node in the tree view you can modify the code or properties of the section represented by the node.

Notice that there is a procedure called htmOffsets in the tree view. When htmOffsets runs in the final Web object, it creates the associations between the HTML form fields and the database fields. The Web object then refers to the offset file, `w-cstget.off`, to determine where to place the data in the HTML output.

Editing process-web-request

The process-web-request procedure is where you add the SpeedScript FIND statement for `w-cstget.w`.

NOTE: Instead of editing process-web-request to add the FIND statement, you could have used the HTML Mapping Wizard to define a query. When you use the wizard, your query is added as an override to the findRecords procedure.

When you view process-web-request in the Section Editor (it is listed under Code Sections in Figure 4–4), you will see that it a runs a number of procedures that control the flow of information between the client and the Web object, and between the Web object and the data source. It also generates header information for the HTML output.

The procedures (assignFields, displayFields, etc.) contain default behavior that you can override. The super procedure, *`install-path`*`/tty/web2/html-map.r`. defines their behavior. You override the default behavior by creating local versions of the procedures. The WebSpeed Agent first looks in the Web Object for a procedure. If the procedure is not found in the Web Object, the Agent looks for it among registered super procedures. For more information about super procedures, see the *Progress Programming Handbook*.

Also note that process-web-request runs a different sequence of processes depending on whether the HTML Form METHOD attribute is GET or POST. With GET, data from the client request is appended to the URL of the Web object. Usually, GET code processes the first request and returns a blank form. With POST, data is sent to the Web object as a separate stream. Usually, POST code processes subsequent requests.

The area in bold text shows where the FIND statement was added to process-web-request in order to create `w-cstget.w`:

w-cstget.w

```
                                    .
                                    .
                                    .

/* STEP 2 -
 * If there are DATABASE fields, find the relevant record that needs to be   *
assigned.
 RUN findRecords.
*/
 FIND Customer WHERE Customer.CustNum eq
 INTEGER(Customer.CustNum:SCREEN-VALUE IN FRAME {&FRAME-NAME}).

                                    .
                                    .
                                    .
```

Notice that the default Run findRecords procedure was commented out.

Compiling and Running

After adding the FIND statement to process-web-request, you can save, compile, and run the Web object from the AppBuilder. Selecting the Save button from the AppBuilder tool bar saves and compiles the file. Selecting the Run button from the AppBuilder runs the Web object (`w-cstget.r`) and displays the output in your default browser.

When you run the Web object, the result looks similar to the Web page shown in Figure 4–5.

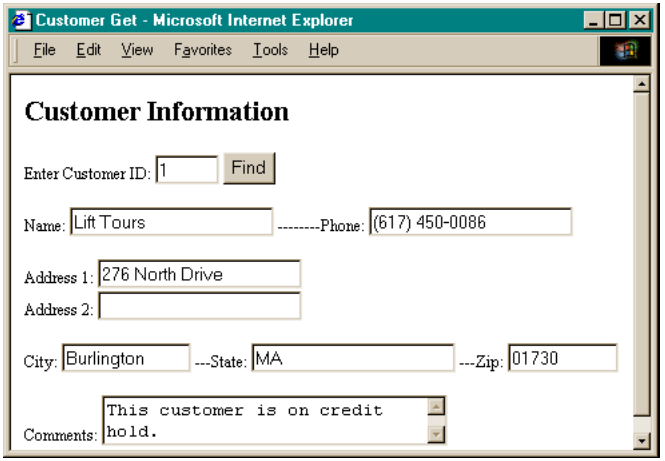


Figure 4–5: Web Page Generated by w-cstget

The Web object displays the first record from the Customer table. When you enter another Customer ID number, it will display the associated record.

Note that unlike `w-sstget.html`, this Web object generates the complete form on the initial request. Having the HTML in a separate file makes any attempt to selectively display parts of the form much more difficult.

SpeedScript Form Buffer

So, how does it work? Here is another look at the SpeedScript added for the Web object in `w-cstget.w`:

```
FIND customer WHERE Customer.CustNum eq
   INTEGER(customer.CustNum:SCREEN-VALUE IN FRAME {&FRAME-NAME}) NO-ERROR.
```

Every Web object that maps to a Web page requires a query mechanism that specifies the data that you want to retrieve from a database. In this case, you use the FIND statement to retrieve the Customer record whose CustNum field (Customer.CustNum) equals the value entered for Enter Customer ID in the Web page. The HTML form element that accepts the entered value is specified by the notation, CustNum:SCREEN-VALUE IN FRAME {&FRAME-NAME}.

This notation identifies a particular SpeedScript field object in the Web object's form buffer (FRAME {&FRAME-NAME}). The *form buffer* is a Web object buffer where all HTML form element values are stored, both for input from and output to a Web page. A *field object* is a SpeedScript construct that defines the visualization of a particular value stored in the form buffer. That is, a field object determines whether the value is represented by a string of

characters, a selection list, a radio set, or some other visual control that is compatible with a corresponding HTML form element. The mapping between a field object and form element is done by a different procedure for each display type. The procedures are located in *install-path*/src/web/support. The procedures used for each display type are defined in a tag mapping file, *install-path*/tagmap.dat, which comes installed with default mappings.

NOTE: The SpeedScript form buffer is identical in function to the Progress 4GL screen buffer. The screen buffer is the buffer from which individual field objects are displayed directly on the screen of a client workstation in a traditional client/server environment. However, SpeedScript access to the client screen is entirely determined by the HTML that it outputs as a block to the Web server. It is essential that you remember this difference when you consult Progress 4GL documentation while doing WebSpeed development. For more information on how input and output differs between the Progress 4GL and SpeedScript, see "SpeedScript and Progress 4GL" in Chapter 5, "SpeedScript."

Using the Automap feature of the AppBuilder, WebSpeed defines database field objects by default for HTML form elements that have the same names. Thus, in this case, the form element name is "CustNum," the database field name. You can also have the AppBuilder map a form element to another database field or SpeedScript variable field object with any valid SpeedScript name up to 32 characters.

The visual characteristics of a field object are defined by properties or attributes associated with that object. In this case, the FIND statement references the SCREEN-VALUE attribute of the CustNum object and is separated from the field object name by a colon (:). The SCREEN-VALUE attribute stores the actual form element/data item value in character string form. For information on setting field object properties and attributes "Customizing Field Object Control Handlers" in Chapter 8, "Controlling WebSpeed Transactions."

SpeedScript also requires each field object to be defined within a special container object known as a *frame*. This SpeedScript frame is not to be confused with HTML frames, but is just another way that SpeedScript organizes field objects in the form buffer. Typically, there is one frame per Web object, identified by the preprocessor name reference, {&FRAME-NAME}, hence CustNum:SCREEN-VALUE IN FRAME {&FRAME-NAME}.

NOTE: When you compile an HTML-mapping Web object in the AppBuilder, it might tell you that it cannot find a particular field object in a frame. You can often resolve this by appending IN FRAME {&FRAME-NAME} to the problem field object reference.

The FIND statement also references the CustNum object as input to the SpeedScript INTEGER function. This function simply converts the numeric string value of the field object to an integer for compatibility with the CustNum database field, which is also an integer.

An Alternative to Form Buffer Input

Here is another way to access HTML input using the FIND statement in `w-cstget.w`:

```
FIND customer WHERE Customer.CustNum eq
  INTEGER(get-value("CustNum")) NO-ERROR.
```

As described previously for embedded SpeedScript Web objects, WebSpeed provides the get-value API function to return values from a Web page. For the CustNum form element, this function accesses the value directly from HTML input. Using this technique avoids any need to reference the form buffer to retrieve input values from the Web.

NOTE: When the POST method is used, process-web-request, calls the inputFields procedure by default. The inputFields procedure loads the form buffer with input by running the get-value function for every field on the form.

4.5.2 Complex HTML Mapping

Figure 4–6 shows an example of an HTML-mapping Web object that is more typical of those that you might find in a full-featured Web application. The Web object, generated from the example code in `w-cstinf.w`, maps to an HTML form with submit buttons and returns customer information from the Sports2000 database.

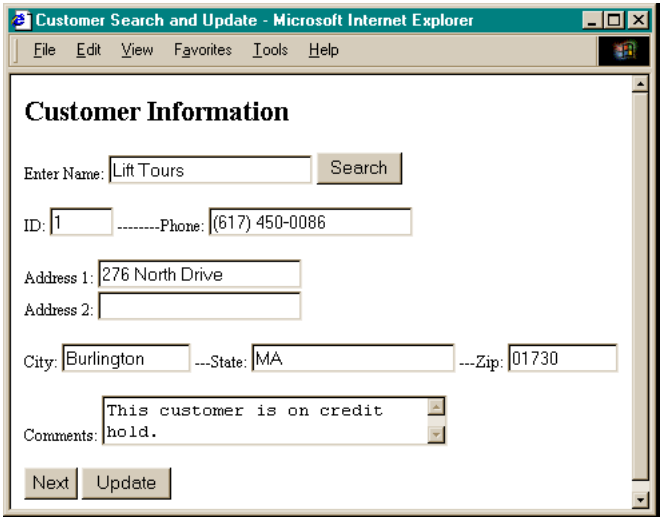


Figure 4–6: Web Page Generated from w-cstinf.w

Basically, the Web object does the following:

- Initially, it generates a Web page similar to Figure 4–6. The Web page contains several buttons and form fields that contain information from the first record (according to the value of CustNum) of the Customer table of the Sports2000 database.
- If you enter a value in the Enter Name field and choose the Search button, the Web object takes the value and finds the customer whose name is equal to or greater than the entered value.

  Thus, entering the value "Dar" causes `w-cstinf.w` to return the first customer whose name begins with "Dar" (which is Dark Alley Bowling).
- The Next button finds and returns the customer record that comes just after the record specified by the current value of the Name element.
- If you change any of the displayed fields, the Update button causes the Web object to update the appropriate customer record.

  After successfully updating the database, the Web object returns the updated Web page with a comment containing today's date appended to the current Comment value. This appended value is also stored in the database for future reference.

The following shows w-custinf.htm, which is the HTML file mapped to w-cstinf.w:

w-custinf.htm

```
<!DOCTYPE HTML PUBLIC "-//Netscape Corp.//DTD HTML
plus Tables//EN" "html-net.dtd">

<HTML>
<HEAD>
<TITLE>Customer Search and Update</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Customer Information</H1>
<FORM ACTION="w-cstinf.w" METHOD="POST">
<P>Enter Name: <INPUT NAME="Name" SIZE="20">
<INPUT TYPE="SUBMIT" NAME="Button Value" VALUE="Search"><BR><BR>
ID: <INPUT NAME="CustNum" SIZE="5">
--------Phone: <INPUT NAME="Phone" SIZE="20"><BR><BR>
Address 1: <INPUT NAME="Address" SIZE="20"><BR>
Address 2: <INPUT NAME="Address2" SIZE="20"><BR><BR>
City: <INPUT NAME="City" SIZE="12">
---State: <INPUT NAME="State" SIZE="20">
---Zip: <INPUT NAME="PostalCode" SIZE="10"><BR><BR>
Comments: <TEXTAREA NAME="Comments" ROWS="2" COLS="30"></TEXTAREA><BR><BR>
<INPUT TYPE="SUBMIT" NAME="Button Value" VALUE="Next">
<INPUT TYPE="SUBMIT" NAME="Button Value" VALUE="Update">
</P>
</FORM>
</BODY>
</HTML>
```

The Web object makes explicit reference to two form elements in this file:

- "CustNum" form field
- "Button Value" submit button

You can follow how the Web object runs by looking at the relevant code sections of the process-web-request procedure for w-cstinf.w.

When the user first goes to the URL for this Web object, `w-cstinf.w` returns the form defined in `w-cstinf.htm` to the browser. It does this by executing the process-web-request procedure. First, the procedure generates the default HTTP header for the Web page that the object is about to return, by calling the default outputHeader procedure (STEP 0 in the code):

w.cstinf.w (RUN outputHEADER)

```
                                .
                                .
                                .

/* STEP 0 -
 * Output the MIME header and set up the object as state-less or
 * state-aware. This is required if any HTML is to be returned to the
 * browser.
 *
 * NOTE: Move RUN outputHeader to the GET section below if you are going
 * to simulate another Web request by running a Web Object from this
 * procedure. Running outputHeader precludes setting any additional
 * cookie information.
 *
 */
RUN outputHeader.
                                .
                                .
                                .
```

It then determines whether the CGI request method is a GET or a POST. In this case, as the first URL request, it is a GET, causing the code to take the ELSE branch of the test:

w-cstinfo.w (GET branch)

```
/* REQUEST-METHOD = GET */
ELSE DO:
                                    .
                                    .
                                    .
  /* STEP 1-
   * If there are DATABASE fields, find the relevant record that needs
   * to be assigned.
   *
  RUN findRecords.
```

The default procedures in this code branch move any initial data to the form buffer, set up the input form elements to receive user input, and send out the prepared Web page.

NOTE: None of this initial GET request requires you to add any additional code to process-web-request. However, you might want to improve the way the Web object fetches the first record. By default, it finds the first record according to the value of CustNum. It might be more useful to display the first customer alphabetically by Name. To change the default behavior, comment out `Run findRecords`. Then, add a line of SpeedScript (for example, `FIND FIRST customer USE-INDEX name.`)

The Web object returns a page to the browser that looks similar to Figure 4–6. The Web page is interactive. The user can view a different record by using the Submit or Next buttons. In addition, the user can change a record by using the Update button. In order to implement this functionality, the following highlighted code is added to the POST branch of process-web-request (recall that POST is specified as the request method in `w-custinf.htm`):

w-cstinf.w (POST branch)

```
  IF REQUEST_METHOD = "POST":U THEN DO:
   vButton = get-value(INPUT "Button Value").

/* STEP 1 -
 * Copy HTML input field values to the Progress form buffer. */
  RUN inputFields.

/* STEP 2 -
 * If there are DATABASE fields, find the relevant record that needs to be   *
assigned.
  RUN findRecords.
*/
  RUN findCustomer.
  IF vButton = "Update" AND AVAILABLE(Customer) THEN
    RUN assignFields.

/* STEP 3 -
 * You will need to refind the record EXCLUSIVE-LOCK if you want to assign   *
database fields below.  For example, you would add the following line.
 *  FIND CURRENT Customer EXCLUSIVE-LOCK NO-ERROR.
*/

  IF vButton = "Update" AND AVAILABLE(Customer) THEN
    RUN assignFields.
```

The first code fragment calls the get-value API function to return a value when the submit button is selected:

```
vButton = get-value(INPUT "Button Value").
```

This function returns the value of any form element or CGI query string element when given the element name. In `w-cstinf.htm`, "Button Value" is the name of all SUBMIT buttons defined in the form (with values "Search", "Next", and "Update"). So, the value of the button that the user pressed to initiate this POST is returned in the variable vButton.

The vButton variable is defined in the definitions section that is global to all procedures in the Web object, `w-cstinf.w`. (Look in the Definitions section of the Section Editor.) This variable has the data type CHARACTER, a variable-length string data type:

```
DEFINE VARIABLE vButton AS CHARACTER NO-UNDO. /* Submit button value */
```

NOTE: The NO-UNDO option is often used to improve SpeedScript memory efficiency. It prevents the automatic saving-to-disk of the specified variable for recovery from a failed database transaction. For more information on database transactions, see Chapter 9, "Controlling Database Transactions."

The form input values sent with the URL is read by calling the default inputFields procedure (STEP 1).

Next, the Customer record based on the value of the SUBMIT button is located. This is done by executing the findCustomer procedure (STEP 2 in the POST branch):

In the code for `findCustomer`, you can see that a newly defined character variable, vName, receives the value of the Name form element entered by the user.

The scope of this variable is local to the findCustomer procedure. However, access to the vButton variable still exists, since it is declared in a section whose scope is the entire Web object:

findCustomer

```
/*------------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:
------------------------------------------------------------------------*/
DEFINE VARIABLE vName AS CHARACTER NO-UNDO.

  vName = get-value("Name"). /* Name value for customer search */

  CASE vButton:
  WHEN "Search" THEN DO WITH FRAME {&FRAME-NAME}:
        FIND FIRST Customer WHERE Name >= vName NO-LOCK NO-ERROR.
        IF NOT AVAILABLE(Customer) THEN
            FIND FIRST Customer USE-INDEX Name NO-LOCK NO-ERROR.
        IF NOT AVAILABLE(Customer) THEN
            Name:SCREEN-VALUE = "*** NO RECORDS ***".
    END.

  WHEN "Next" THEN DO WITH FRAME {&FRAME-NAME}:
        FIND FIRST Customer WHERE Name > vName NO-LOCK NO-ERROR.
        IF NOT AVAILABLE(Customer) THEN
            FIND FIRST Customer USE-INDEX NAME NO-LOCK NO-ERROR.
        IF NOT AVAILABLE(Customer) THEN
            Name:SCREEN-VALUE = "*** NO RECORDS ***".
    END.

  WHEN "Update" THEN DO WITH FRAME {&FRAME-NAME}:  /* CustNum to update */
        FIND Customer WHERE CustNum = INTEGER(get-value("CustNum"))
            EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
        IF LOCKED(Customer) THEN
            Name:SCREEN-VALUE = "*** LOCKED ***".
        ELSE IF NOT AVAILABLE(Customer) THEN
            Name:SCREEN-VALUE = "*** NOT FOUND ***".
        ELSE
            Comments:SCREEN-VALUE =
                Comments:SCREEN-VALUE +
                "~n*** Updated " + STRING(TODAY) + " ***".
    END.
  END CASE.
END PROCEDURE.
```

The code first returns the value of the Name form element from the get-value API function. Then it checks for the three possible values of the Submit button returned in process-web-request. This is done using a CASE statement. It executes the WHEN option that specifies the current value of the CASE statement expression, in this case, vButton.

In the case of "Search", the first customer record whose Name field is equal to or greater than the value of the Name form element is found. If such a record does not exist, it returns the first customer record to restart the search. If no customers are on file, it returns a suitable message in the Name form element.

NOTE: The WITH FRAME {&FRAME-NAME} option indicates that all data item references in the block (in this case a DO block) that require a parent SpeedScript frame default to {&FRAME-NAME}. This is a shortcut to avoid specifying a frame for each data item reference.

In the case of "Next", the processing is almost the same as for "Search". Assuming that the user has already seen this record, it searches for the record with the next greater value for Name. Again, if no such record exists, it returns the first record, and if no customers are on file it returns a suitable message.

NOTE: The current value of Name is being used as if it were the name that the user last requested. However, there is no guarantee of this, since the user could change the value before submitting the form. If this Web object were state aware (part of a WebSpeed transaction), you could replace the FIND FIRST Customer statement with a FIND NEXT Customer statement that would automatically find the next Customer record after the “current” one in the database. This guarantees that the application returns the next record found and not the one based on the returned value of the Name form element. In a stateless Web object, there is no previous record, since it always executes as if for the first time. Thus, the entered value of Name must be used. (You could also use a hidden HTML form element to duplicate the returned value of the database Name field for later reference.) The Web object examples do include a state-aware version of this Web object, `w-cststa.w`, which you can compare to `w-cstinf.w`. For more information on building state-aware Web objects, see Chapter 8, “Controlling WebSpeed Transactions.”

For "Update", you must find a unique record that corresponds to the returned values, since there can be duplicate customer names. The CustNum field is the primary key for Customer records, you can assume that the value passed back in the CustNum form element specifies the record intended for update. (However, the user could have changed it intentionally or accidentally.) If there is no such record, or the specified record is locked by another Web client, the form element Name value is replaced with a suitable message.

If the record does exist, a confirmation is concatenated (+) to the Comments form element value, both for immediate confirmation and for future reference. The SpeedScript TODAY function returns today's date, and the STRING function converts the DATE value (a separate data type) to a character string for the concatenation. Also, ~n specifies a newline in a string.

NOTE: The FIND statement for the "Update" case specifies that the retrieved record be locked (EXCLUSIVE-LOCK). This prevents other Web clients from updating the record while the current update takes place. For more information on record locking in SpeedScript, see the *Progress Programming Handbook*.

At this point, the process-web-request method procedure has either found the record or has not found the record that the user intended to update. The procedure returns from findCustomer and executes the following code from w-cstinf.w (POST branch):

```
/* STEP 3 -
 * Assign the fields from the form buffer to the database. */
IF vButton = "Update" AND AVAILABLE(Customer) THEN
  RUN assignFields.
```

The test of vButton is added, again, because the next default event procedure, assignFields, assumes that you want to update the current record with the values received from the user. However, this is only true if the user submitted the form for update, and only if the specified record was actually found. The SpeedScript AVAILABLE function returns a logical value of TRUE if there is a record in the buffer for the specified database table, in this case, Customer.

The remaining default event procedures in the POST request branch move the field values from the current (possibly updated) record to the form buffer, prepare any form elements for further input from the user, and output the Web page with the resulting content to the browser.

4.5.3 Complex HTML Mapping That Includes a SmartDataObject

Figure 4–7 shows the Web page generated by another sample WebSpeed application, `w-custdir.w`.

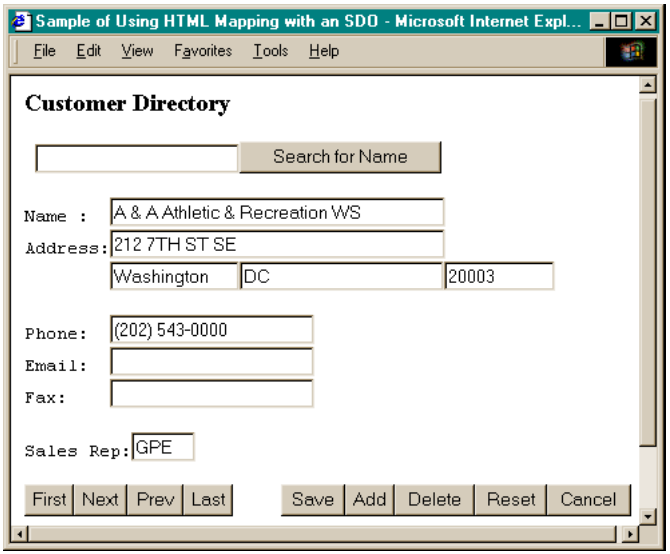


Figure 4–7: Web Page Generated from w-custdir.w

The Web page shown in Figure 4–7 interacts with the Customer table of the Sports2000 database like the other examples in this section. However, it also has the ability to add or change records in the table. Moreover, it uses a SmartDataObject to define its query logic. The following sections describe these features in more detail.

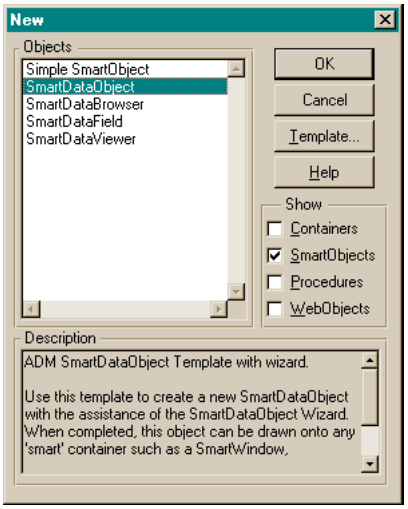
Creating a SmartDataObject

SmartDataObjects provide query logic that is separate from the rest of your application. As separate modules, they can be used by multiple applications, can be refined at run time, and can be modified more easily than queries that are embedded within applications. Another advantage to SmartDataObjects is that they can be used on an AppServer for multi-tier deployment. See *Building Distributed Applications Using the Progress AppServer* for more information.

SmartDataObjects can easily be created in the AppBuilder. The SmartDataObject associated with `w-custdir.w` is called `dcustomer.w`. You can create it by performing the following steps:

1 ♦ Choose File→ New from the AppBuilder main menu.

The New Object dialog box appears:



2 ♦ Select SmartDataObject from the list of SmartObjects.

Note that AppBuilder must be connected to a database in order to create a SmartDataObject. The procedures for starting and connecting to a database are described in *Getting Started with WebSpeed*.

3 ♦ Choose Next on the SmartDataObject Wizard, then choose Define Query from page 2 of the wizard.

4 ♦ In the Query Builder:

a) Add the Customer table to the Select Tables list.

b) Select the Sort button.

c) Add Name to the Selected Fields list.

5 ♦ Go to the next dialog box in the wizard (page 4) and choose Add Fields.

6 ♦ Choose Add and add all of the available fields to the Selected Fields list.

7 ♦ Go to the final page of the wizard and choose Finish.

You will see a window labeled Untitled, which represents the SmartDataObject you just created. You can save it as `dcustomer.w` from the File menu of the AppBuilder.

When you save a SmartDataObject, you create a number of files: .w, .r, _cl.w, _cl.r, and .i. For more information about the function of these files, or for more information about SmartDataObjects in general, see the *Progress ADM 2 Guide* and the *Progress ADM 2 Reference*.

For more information about creating SmartDataObjects with the AppBuilder, see the *Progress AppBuilder Developer's Guide*.

Mapping With a SmartDataObject

You can specify dcustomer.w as the data source when you use the HTML Mapping Wizard to map an HTML file like w-custdir.htm:

w-custdir.htm

```
<html>
<head>
<title>Sample of Using HTML Mapping with an SDO</title>
</head>
<h3><font color="#000000">Customer Directory</font></h3>
<form ACTION="w-custdir.w" METHOD="POST">
<input type="hidden" name="cusrowid" >
<input type="hidden" name="AddMode" >
<pre>
<input NAME="SearchName" SIZE="20">
<input TYPE="submit" NAME="requestedAction" VALUE="Search for Name">
Name :  <input NAME="Name" SIZE="34">
Address: <input NAME="Address" SIZE="34">
        <input NAME="City" SIZE="12">
        <input NAME="State" SIZE="20">
        <input NAME="PostalCode" SIZE="10">
Phone:  <input NAME="Phone" SIZE="20">
Email:  <input NAME="EmailAddress" SIZE="20">
Fax:    <input NAME="Fax" SIZE="20">
Sales Rep:<input NAME="SalesRep" SIZE="5">
<input TYPE="submit" NAME="requestedAction" VALUE="First">
<input TYPE="submit" NAME="requestedAction" VALUE="Next">
<input TYPE="submit" NAME="requestedAction" VALUE="Prev">
<input TYPE="submit" NAME="requestedAction" VALUE="Last">
<input TYPE="submit" NAME="requestedAction" VALUE="Save">
<input TYPE="submit" NAME="requestedAction" VALUE="Add">
<input TYPE="submit" NAME="requestedAction" VALUE="Delete">
<input TYPE="submit" NAME="requestedAction" VALUE="Reset">
<input TYPE="submit" NAME="requestedAction" VALUE="Cancel">
</pre>
</form>
</body>
</html>
```

You can use the HTML Mapping Wizard's Automap feature to map the fields referenced in w-custdir.htm to the database fields referenced in dcustomer.w. Note that all the fields will map except the hidden fields and the SearchName field (shown in bold). These fields are used to pass information to the WebSpeed Agent and do not reference fields in the database.

The hidden field, cusrowid, holds a value that identifies the current record that is displayed on the Web page. It establishes a context for the WebSpeed Agent when the next request is submitted. The other hidden field, AddMode, holds a value that tells the WebSpeed Agent if a new record is to be added or not when the next request is submitted. The code that utilizes these hidden fields is explained in the following section.

Adding Code

After the HTML Mapping Wizard completes, you will see a tree view of an untitled Web object, which is similar to Figure 4–8.

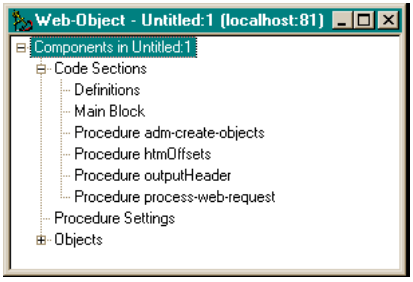


Figure 4–8: Tree View of an Untitled HTML-mapping Object

From the Code Sections branch of the tree view, you can add all the code that is necessary to complete `w-custdir.w`. With Code Sections selected, choose the Edit Code button from the AppBuilder Tool Bar to start the Section Editor. The Section Editor allows you to add new procedures, to modify default procedures, and to invoke and override super procedures. For more information, see the *Progress AppBuilder Developer's Guide*.

You will need to do the following in order to complete the Web object:

1. Create a trigger that responds to the value of cusrowid.

   In the trigger, the value of cusrowid is passed to the setCurrentRowids procedure. Recall that the value of cusrowid is stored in a hidden field and is set in a prior WebSpeed transaction. The trigger essentially tells the WebSpeed Agent which row in the table to refer to. In other words, the trigger establishes a context for the WebSpeed Agent by passing information about a previous transaction.

   In the example Web object, the trigger is called Web.input and the code looks similar to the following:

Web.input

```
/*------------------------------------------------------------------------
  Purpose:     Assigns form field data value to frame screen value.
  Parameters:  p-field-value
  Notes:       This input trigger of the hidden field cusrowid sets the
               current row to the one in the hidden field to set the
               context in preparation for further processing.
------------------------------------------------------------------------*/
  DEFINE INPUT PARAMETER p-field-value AS CHARACTER NO-UNDO.

  DO WITH FRAME {&FRAME-NAME}:
    setCurrentRowids (p-field-value).
  END.

END PROCEDURE.
```

2. Create a procedure that responds to the AddMode flag.

   In the example Web object the procedure is called inputFields and the code looks similar to the following:

inputFields

```
/*------------------------------------------------------------------------
  Purpose:     Super Override
  Parameters:
  Notes:  After standard behavior for this procedure, this procedure
  override looks at a hidden field called AddMode, and sets a flag
  indicating that this record is new, and must be created in the
  database before the fields are assigned.  The AddMode flag is then
  turned off in preparation for sending the next page.
------------------------------------------------------------------------*/
 RUN SUPER.

 IF  ab_unmap.AddMode:screen-value in frame {&FRAME-NAME} = "YES" THEN
 DO:
  PleaseAdd = TRUE.
  setAddMode (TRUE).
 END.
 IF  ab_unmap.AddMode:screen-value in frame {&FRAME-NAME} = "NO" THEN
 DO:
   PleaseAdd = FALSE.
   setAddMode (FALSE).
 END.
 ab_unmap.AddMode:screen-value in frame {&FRAME-NAME} = "No".

END PROCEDURE.
```

As noted in the comment, this procedure is actually an override to the inputFields super procedure. This override supplements the standard behavior of inputFields by allowing it to react to the value of the AddMode flag in the hidden field. This code allows the Web object to add a new record to the database.

3. Create a procedure that manages the unmapped fields.

   In the example Web object, the procedure is called outputFields and the code looks similar to the following:

outputFields

```
/*------------------------------------------------------------------------
  Purpose:     Super Override
  Parameters:
  Notes: This outputs the current record to the hidden field cusrowid
  before standard behavior for this procedure and outputs the AddMode
  hidden field.
------------------------------------------------------------------------*/
IF getUpdateMode() = "Add" THEN
    ab_unmap.AddMode:screen-value in frame {&FRAME-NAME} = "YES".
ELSE
    ab_unmap.AddMode:screen-value in frame {&FRAME-NAME} = "NO".

ab_unmap.cusrowid:screen-value in frame {&FRAME-NAME} = getRowids().

RUN SUPER.

END PROCEDURE.
```

As noted in the comment, this procedure is actually an override to the outputFields super procedure. Its purpose is to identify the correct row for the next transaction. This override sets the value of ab_unmap.custrowid to the rowid of the current record. The value of ab_unmap.custrowid is set before running outputFields.

The ab_unmap temporary table holds the values of the fields that are unmapped. In this example, the unmapped fields are cusrowid, SearchName, and AddMode.

NOTE: The AppBuilder online help contains reference pages that describe the syntax and behavior inputFields, outputFields, and all other super procedures that apply to WebSpeed. For more information about using super procedures, see the *Progress ADM 2 Guide* and the *Progress ADM 2 Reference*.

4. Modify process-web-request so that the Web object responds to button events.

   The button events are handled in a case statement, as shown in the following segment:

process-web-request

```
IF REQUEST_METHOD = "POST":U THEN DO:
  RUN inputFields.
  RUN findRecords.
  CASE get-field ("requestedAction"):
      WHEN "First" THEN
        RUN fetchFirst.
      WHEN "Next" THEN
        RUN fetchNext.
      WHEN "Prev" THEN
        RUN fetchPrev.
      WHEN "Last" THEN
        RUN fetchLast.
      WHEN "Search for Name" THEN DO:
        addSearchCriteria('name',get-value('searchname')).
        RUN findRecords.
      END.
      /* Maintenance action selected  */
      WHEN "Save" THEN DO:
          IF getUpdateMode () <> "add" THEN
        RUN fetchCurrent.
        RUN assignFields.
        setAddMode (FALSE).
        setUpdateMode ("").
      END.
      WHEN "Delete" THEN DO:
        RUN fetchCurrent.
        deleteRow().
      END.
      WHEN "Reset" THEN DO:
        RUN fetchCurrent.
      END.
      WHEN "Cancel" THEN DO:
        RUN fetchCurrent.
        setUpdateMode ("").
      END.
      WHEN "Add" THEN DO:
        RUN fetchCurrent.
        setUpdateMode ("Add").
      END.
  END CASE.
```

5

SpeedScript

This chapter contains information about SpeedScript and embedded SpeedScript including the following:

- An overview of SpeedScript and Progress 4GL
- Elements of SpeedScript syntax
- WebSpeed Preprocessors
- APIs for message handling
- APIs for passing information between web requests
- Miscellaneous API functions
- XML support in SpeedScript
- An overview of Embedded SpeedScript
- Embedding SpeedScript in HTML
- Building Interactive Web Pages with embedded SpeedScript Files
- Using <META> and <!--WSMETA--> Tags
- Developing embedded SpeedScript Web Objects

5.1 SpeedScript and Progress 4GL

SpeedScript is an implementation of the Progress 4GL language that is primarily used to develop Web applications. The documentation available for the 4GL language is, for the most part, applicable to SpeedScript. See the following for more information regarding SpeedScript:

- *Progress Language Reference*—Describes the syntax of all SpeedScript language elements. Also identifies which language elements apply to 4GL, to SpeedScript, or to both 4GL and SpeedScript.
- *Progress Programming Handbook*—Covers important programming concepts such as database locking and transaction rules, program block and resource scoping rules, and the use of persistent procedures.

5.1.1 Similarities

The similarities between 4GL and SpeedScript include:

- Block structure and resource scoping rules are the same.
- Database events (such as CREATE, and DELETE) can be handled in the same way.
- Database locking and database transaction rules are the same.
- Both can use SmartDataObjects as data sources.
- Both can use the AppServer to process requests.
- Both can be written using the same AppBuilder tools (Procedure Window, the Section Editor and the TreeView).

5.1.2 Differences

Some important differences between SpeedScript programming and Progress 4GL programming are:

- 4GL applications are usually state-aware, while SpeedScript applications are often stateless. The distinction between state-aware and stateless applications is discussed in Chapter 8, "Controlling WebSpeed Transactions."
- GUI widget events are not used in WebSpeed SpeedScript applications. Visual elements are handled by HTML, rather than as GUI widgets.
- The preprocessor {&OUT} statement is used to output data to the HTML page, rather than the DISPLAY statement.

- In SpeedScript, all terminal-focused I/O is replaced by a block mode Web-oriented I/O, where the SpeedScript frame is the block data structure. Therefore, I/O blocking statements such as UPDATE or PROMPT-FOR cannot be used in SpeedScript applications. Similarly, the I/O blocking options of statements like the MESSAGE (VIEW-AS ALERT-BOX) and PAUSE (MESSAGE) statements are ignored.

- Very few Progress 4GL events apply to WebSpeed applications, except database events. The one essential event in SpeedScript is WEB-NOTIFY. However, in normal use, this event is handled exclusively by the Agent control program (`web-disp.p`).

- WAIT-FOR cannot be used for user input in SpeedScript applications.

- Some procedures that are available through include files and the "Insert Call" button of the Section Editor are only appropriate for WebSpeed applications. Some of these are:

  - getWebState
  - getWebTimeRemaining
  - hidden-field-list
  - hidden-field-list
  - set-cookie
  - get-cookie

  Two files, `admweb.i` and `html-map.i`, contain the calls that are only appropriate in a WebSpeed application. A list of these calls can be found in the source files themselves.

- SpeedScript includes special extensions, including a virtual Web output device ("WEB") to define Web page output streams to your Web server and the WEB-CONTEXT system handle to access the request environment. However, most of these extensions are wrapped in the API functions, method and event procedures, and preprocessor definitions provided with WebSpeed.

These examples also rely on SpeedScript preprocessor references, especially {&OUT} and {&DISPLAY}, to direct output to the WebSpeed-defined output stream, WebStream. You can find the definitions for these preprocessor references (and several others) in *install-path*`/src/web/method/cgidefs.i`. For more information on the {&DISPLAY} preprocessor reference, see Chapter 6, “Handling DISPLAY Output.” For more information on the SpeedScript preprocessor and more advanced SpeedScript topics, see the *Progress Programming Handbook*.

5.2 Elements of SpeedScript Syntax

SpeedScript is a block-structured, but statement-oriented language. That is, much of the behavior of a WebSpeed application depends on how statements are organized into blocks, but the basic executable unit of a WebSpeed application is the statement.

Statements

This is the basic syntax of a WebSpeed application:

SYNTAX

```
statement { . | : } [ statement { . | : } ...
```

Thus, a SpeedScript application consists of one or more statements. Each *statement* consists of a number of words and symbols entered in a statement-specified order and terminated by a period (.) or a colon (:), depending on the type of statement.

NOTE: SpeedScript is a case-insensitive language. However, by convention, SpeedScript keywords (such as FOR) are expressed in uppercase lettering. Also by convention, most preprocessor names in WebSpeed are expressed in uppercase and the CGI environment variables defined in SpeedScript are expressed in uppercase according to the usual CGI conventions. However, character string compares and database queries keyed on character strings can be case-sensitive or case-insensitive, depending on the context and specifications of the compare or query.

`sample1` is a one-statement application that displays "Hello, World!" in a browser:

sample1

```
{&OUT} "<H1> Hello, World! </H1>" .
```

NOTE: You can experiment with the sample code in this section by running the code in the WebTools Scripting Lab. From the AppBuilder, choose Tools→ WebTools and then choose Scripting Lab from the WebTools menu.

Comments

A WebSpeed application can also contain nonexecutable comments wherever you can put white space (except in quoted strings). Each comment begins with "/*" and terminates with "*/", and you can nest comments within other comments.

Blocks

In SpeedScript, a *block* is a sequence of one or more statements, including any nested blocks, that share a single context. A *context* consists of certain resources that a block of statements share. The content of this shared context depends on the type of block and its relationship to other blocks. `sample2` shows a typical layout of blocks in a procedure:

sample2

```
REPEAT WHILE TRUE: /* BEGIN Iterative Block */
    RUN max-customers.
END.              /* END Iterative Block */

PROCEDURE max-customers: /* BEGIN Internal Procedure Block */
    FOR EACH Customer USE-INDEX Name NO-LOCK: /* BEGIN Iterative Block */
        IF Balance > 20000 THEN DO: /* BEGIN Non-iterative Block */
            {&DISPLAY} Name Balance.
        END.                          /* END Non-iterative Block */
    END.                                      /* END Iterative Block */
END PROCEDURE.          /* END Internal Procedure Block */
```

The most basic block is the procedure, and the most basic procedure is an external procedure-a file containing one or more statements-because this is the smallest unit that WebSpeed can compile separately. Web objects are always external procedures. Most of the samples in this section are not external procedures because their execution depends on the context of an enclosing Web object. An external procedure block is also the only type of block that requires no special syntax to define it. WebSpeed always defines an external procedure block, by default, when the procedure executes.

You must begin all other types of blocks with appropriate header statements. Header statements, such as the DO, FOR, and PROCEDURE statements shown in `sample3`, are typically terminated with a colon, although you can use a period. Generally, you must terminate the block begun with a header statement with an END statement.

5.2.1 Running Procedures and Web Objects

The statement that executes a procedure in SpeedScript is the RUN statement. This statement has several forms, depending on the application. The most basic executes a procedure by name. If the named procedure is defined internally to the executing procedure file (or Web object), SpeedScript executes that internal procedure. Otherwise, it looks for an external procedure file to execute.

NOTE: In SpeedScript, an external procedure file is any legal set of SpeedScript statements in a file. The PROCEDURE keyword is needed only to define internal procedures.

You can also pass parameters to procedures that define them using DEFINE PARAMETER statements. For complete information on the options available for the RUN statement, see the *Progress Language Reference*.

Persistent Procedures

Among the most important forms of the RUN statement is the RUN statement using the PERSISTENT option. When you execute an external procedure, the PERSISTENT option causes the procedure context, including all of its data and internal procedure definitions to remain active in memory after the main-line of the procedure has returned to the caller.

This feature allows you to create (instantiate) SpeedScript objects, of which Web objects are the most important in WebSpeed. This is possible because when you RUN a procedure persistently, you can obtain a handle to the procedure's active context. This procedure handle allows you to execute any internal procedure defined inside the persistent procedure context. You do this using the IN option of the RUN statement, where you run an internal procedure "IN" the handle of the persistent procedure. Thus, the persistent procedure maintains state for all of its internal procedure executions.

This is also how method and event procedures are implemented in WebSpeed. Generally, you run method procedures in the handle web-utilities-hdl (for the common utility method procedures in `web-util.p`) or the handle THIS-PROCEDURE (which references the current external procedure or Web object).

WebSpeed Calling Conventions

WebSpeed has a special set of procedure calling conventions. The first convention relies on the run-web-object method procedure. This procedure is the standard method to execute a Web object from within another procedure. It is also the basic method `web-disp.p` uses to execute Web objects in response to Web requests. The run-web-object procedure follows a protocol designed to ensure the integrity of the Web object, whether it is stateless or state aware. This allows WebSpeed to manage Web objects in a consistent manner. The run-web-object method procedure is defined in the utility object, *`install-path`*`/src/web/objects/web-util.p`.

The second convention relies on the dispatch method procedure. You use this procedure to execute event procedures, the special class of method procedures that you can override, but whose default code you cannot ordinarily change directly. The dispatch method procedure ensures that the override is executed rather than the default if the override exists. The dispatch method procedure is defined in the include file, *install-path*/src/web/method/admweb.i.

5.2.2 Blocks and Context

The context of a block generally lies within the code defined by the beginning and end of the block. For an external procedure, the block beginning and end is the first and last statement in the file. For any other block, it is the block's header and END statement. The context of an external procedure includes all SpeedScript data, objects, triggers, and internal procedures that it defines. This is the only type of block that can define both trigger and internal procedure blocks within its context. Thus, the external procedure context is often called the main block of the procedure. For example, the variable, balance-sum, is defined in the main block of `sample3` and can be modified by all statements and blocks defined within `sample3`.

NOTE: A SpeedScript trigger is a statement or block of code that executes in response to a SpeedScript event and is usually specified with an ON statement. SpeedScript events have limited but important uses in WebSpeed. They are identified by event keywords that include WEB-NOTIFY, CLOSE, and several database events. For more information on the ON statement and other trigger statements, see the *Progress Programming Handbook* or the *Progress Language Reference*.

In general, any data or objects that you define within the context of a procedure are available only to the statements of that procedure. However, any data or objects that you define within other types of blocks are actually part of the nearest enclosing procedure context, not the context of the block where they are defined.

For example, this procedure calculates the sum of customer balances using a variable, balance-sum, defined within a FOR block:

sample3

```
FOR EACH Customer FIELDS (Balance):
    DEFINE VARIABLE balance-sum AS DECIMAL INITIAL 0.
    balance-sum = balance-sum + Balance.
END.

{&OUT} "<P>" "Corporate Receivables Owed:"
        STRING(balance-sum, "$>,>>>,>>>,>>9.99") "</P>" .
```

However, balance-sum is actually assigned to the outer procedure context, so the {&OUT} statement, outside the FOR block, can also reference it.

Variable definitions are always assigned to the nearest enclosing procedure or trigger context, because WebSpeed maintains the run-time values for variables in a single stack frame for the entire procedure. Thus, when a procedure A calls another procedure B, all of procedure A's variable values are saved for the return from procedure B. However, when WebSpeed executes a DO, FOR, or other type of block, no prior variable values are saved before the block executes. Likewise, there is no separate data context to maintain variables defined within these blocks. Therefore, WebSpeed maintains the variables defined within these blocks as though they were defined directly in the procedure context.

5.2.3 Block Context and Resource Scope

The context of some blocks also helps determine the scope of certain resources. Conversely, the scope of other resources might have little to do with the context in which you initially define them. *Scope* is really the duration that a resource is available to an application. Scope can vary depending on the resource and the conditions of application execution.

In general, the scope of resources created at *compile time* (when WebSpeed compiles your application) is determined at compile time; the scope of resources created at *run time* (when WebSpeed executes your application) is determined at run time. (See the "Compile-time Versus Run-time Code" section.) The scope of a resource begins when the resource is instantiated (created in your application) and ends when the resource is destroyed (removed from your application).

For example, a FOR statement defines the scope of any database buffer that it implicitly defines for record reading. The scope of such a record buffer is identical to the context of the FOR block, because the buffer is deleted when the FOR block finishes with it. For example, in `sample4`, the scope of the Customer buffer ends when the FOR block completes. Although the {&DISPLAY} statement following the FOR block can access balance-sum, it can no longer access the Balance field for any record read into the Customer buffer by the FOR block.

Unscoped Resources

Note that `sample4` does not compile if executed from an HTML-mapping process-web-request procedure, because frames defined in an external procedure (Web object) do not scope to any of its internal procedures. As a result, the frame is invisible to the statements in the FOR block and the field object references, which are assumed to be in the Web object frame (FRAME {&FRAME-NAME}), have no recognized existence:

sample4

```
FOR EACH sports.Customer
  FIELDS (sports.Customer.Name sports.Customer.Balance):
    DEFINE VARIABLE balance-sum AS DECIMAL INITIAL 0.
    balance-sum = balance-sum + Balance.
    {&DISPLAY} sports.Customer.Name sports.Customer.Balance.
END.

FIND FIRST sports.Customer.
{&DISPLAY} sports.Customer.Name sports.Customer.Balance
  WITH FRAME {&FRAME-NAME}.
```

The solution is to copy the WITH FRAME {&FRAME-NAME} reference to just before the colon terminator of the FOR EACH statement.

Dynamic Resources

In general, dynamic resources are resources that you implicitly or explicitly create and delete at run time. Record buffers that a FOR statement implicitly defines are dynamic buffers. WebSpeed creates them when the FOR block executes and deletes them when its execution completes. SpeedScript frames scoped to a FOR block are dynamic in the same way. The whole context of a procedure is dynamic. WebSpeed creates its local data resources when you call the procedure and deletes them when the procedure returns or otherwise goes out of scope.

However, WebSpeed allows you to create and delete certain dynamic resources explicitly. For WebSpeed, this especially applies to the external procedure contexts (persistent procedures) of Web objects. In general, the scope of a dynamic resource lasts from the time you create it to the time you delete it or when the WebSpeed Agent session ends, whichever occurs first. When a procedure block completes execution and its context goes out of scope, this does not affect the scope of any dynamic resources that you have created in that context. The completion of the procedure can only affect whether the resources are still accessible from the context that remains.

WebSpeed allows you to define handles to most dynamic resources. These handles are variables that point to the resources you have created. You must ensure that handles to these resources are always available to your application until you delete the resources. If you create a persistent procedure context and do not explicitly delete it, the context remains in scope for the duration of the Agent session, no matter what Web client accesses it. If you also lose the original handles to these procedure contexts, you might not be able to access them again. If you cannot access them, you cannot delete them, and in effect, they become memory lost to WebSpeed and the system (that is, a memory leak).

5.2.4 Compile-time Versus Run-time Code

Like many languages, SpeedScript includes two basic types of code:

- Compile-time, sometimes known as nonexecutable code
- Run-time, sometimes known as executable code

However, as an interpretive language, WebSpeed syntax combines compile-time and run-time components in many more ways than a compiled language like C. The flexibility of this syntax helps implement the rich variety of overridable defaults that characterizes SpeedScript.

Compile-time Code

Certain statements exist only to generate r-code when WebSpeed compiles them. These are *compile-time* statements. That is, they create static data and form buffer resources (frame and field objects) that the run-time statements can reference and modify, but not destroy, during execution.

Compile-time Syntax Elements

Most compile-time code consists of the following syntax elements:

- Compile-time statements, including SpeedScript statements that begin with the DEFINE keyword
- The nonexecutable components of block header statements and END statements
- Options and phrases associated with compile-time statements, wherever they appear
- Literal expressions (constants)
- Preprocessor directives and definitions

Run-time Code

Run-time statements use the static resources created by compile-time statements, but can also create, use, and destroy dynamic resources at run time. That is, run-time statements include statements that interact with static resources, dynamic resources, or both. Many run-time statements also include compile-time options. These are options that generate resources at compile time that are later used by the same statements at run time.

Run-time Syntax Elements

Most run-time code consists of the following syntax elements:

- All statements other than compile-time statements.
- The options and phrases associated with run-time statements, including the executable components of block header statements, **except** those options and phrases that are also associated with compile-time statements (such as Format and Frame phrases).

  The block header statements of iterative blocks (DO, FOR, and REPEAT blocks) all have executable components. These executable components monitor and enforce the iteration conditions (when the block starts and stops executing) and select any database data that are available to the block.
- Assignment statements and nonliteral expressions (variables, functions, attributes, and methods).

  NOTE: WebSpeed distinguishes variables from their field object representation. However, the references to variables (or fields) in the ASSIGN and {&DISPLAY} statements reference both the variables and their field object representations because WebSpeed moves data between them implicitly at run time.

How Do Compile-time and Run-time Code Interact?

Because SpeedScript is a run-time interpreted language, it can combine compile-time and run-time code in a number of interesting and powerful ways.

As noted earlier, some run-time statements can also include compile-time options. Thus, you can define a frame to display data using a DEFINE FRAME statement, then add options to that static definition using Frame phrase options in subsequent run-time statements, such as FOR and {&DISPLAY}.

In this example, the data fields, frame type, and title for frame alpha are all defined at compile time and in three different statements:

sample5

```
DEFINE FRAME alpha Customer.Name Customer.Phone.

FOR EACH Customer WITH FRAME alpha SIDE-LABELS:
    {&DISPLAY} name phone balance WITH TITLE "Customer Balances".
END.
```

A powerful example of the interaction between compile-time and run-time code is the use of the VALUE option in a number of run-time statements. In `sample6`, the VALUE option allows you to use a run-time expression (proc-name[proci]) to provide a compile-time object name:

sample6

```
DEFINE VARIABLE proci AS INTEGER.
DEFINE VARIABLE proc-name AS CHARACTER EXTENT 3
    INIT ["proc1.p", "proc2.p", "proc3.p"].

{&OUT} "<P>These are STATIC procedure executions.</P>" .
RUN proc1.p.
RUN proc2.p.
RUN proc3.p.

{&OUT} "<P>These are DYNAMIC procedure executions.</P>" .
DO proci = 1 TO 3:
    RUN VALUE(proc-name[proci]).
END.
```

In the RUN statement, the object name is the name of a procedure to execute. `sample7` thus shows how the same three procedures can be executed using static compile-time object names or using object names evaluated by the VALUE option at run time.

NOTE: The procedures `proc1.p`, `proc2.p`, and `proc3.p` exist for illustration only.

5.3 WebSpeed Preprocessors

The preprocessor is a function of the Progress 4GL compiler that also applies to SpeedScript. On its initial pass through source code, the compiler looks for preprocessor directives and performs text substitutions when it finds them. All directives begin with an ampersand (&).

The WebSpeed preprocessors, which are listed in Table 5–1, provide consistent access to the Web environment, especially the Web output stream. The definitions of WebSpeed preprocessor names reside in *install-path*/src/web/method/cgidefs.i.

Table 5–1: WebSpeed Preprocessors

Preprocessor Name	Assigned Value
&WEBSTREAM	STREAM Webstream
&OUT	PUT {&WEBSTREAM} UNFORMATTED
&OUT-FMT	PUT {&WEBSTREAM}
&DISPLAY	DISPLAY {&WEBSTREAM}

5.4 WebSpeed API Functions

This section describes some of the commonly used WebSpeed API functions. For more information on WebSpeed API functions, see the AppBuilder online help. Choose Help→ Help Topics from the AppBuilder menu bar. Then select the Find tab and enter WebSpeed API in the top field of the dialog box.

5.4.1 Message Handling

A number of WebSpeed API functions facilitate message queuing and output. They interact with a message queue that allows you to organize messages into named groups. Table 5–2 lists these functions.

Table 5–2: Message API Functions

Function	Return Type	Description
available-messages	LOGICAL	Returns TRUE if there are any messages queued for a specified group or for all groups.
get-messages	CHARACTER	Returns any messages queued for a specified group or for all groups. Optionally deletes the messages from the queue.
get-message-groups	CHARACTER	Returns a comma-separated list of groups for which there are queued messages.
output-messages	INTEGER	Outputs messages to the Web that have been queued by queue-message and returns the number of messages output. Includes options to format the output and to specify a specific group of messages or all messages.
queue-message	INTEGER	Queues a message for later output by output-messages and returns the message number in the queue. Optionally associates the message with a specified group.

The definitions for these functions reside in *install-path*`/src/web/method/message.i`.

Generating Messages Directly

In embedded SpeedScript and SpeedScript-generating Web objects, where the program code explicitly controls the HTML output, you can call the message API function directly. For most applications you only need to use output-messages and queue-messages. These API functions call the other message API functions.

Generating Messages with a Custom Tag

For HTML-mapping Web objects, WebSpeed provides a custom tag (<!--WSMSG -->) in `tagmap.dat` to output messages at strategic points in a mapped HTML file. Generally, you invoke the queue-message function to append a message to the message queue. If you want a certain group of messages to be output to the Web, you can insert the <!--WSMSG --> tag for that group of messages at any point in the HTML file.

This custom tag takes two optional attributes:

- NAME—The name of the message group or "all" for all messages in the queue
- VALUE—The heading that is to appear prior to the messages output for the specified group

Using the custom tag with no attributes defaults to NAME="all" and TYPE="application messages".

The following HTML example contains two <!--WSMSG --> tags, one for messages in a "CustNum" group and one for messages in a "CustName" group. When the HTML-mapping Web object invokes the output-fields event procedure, the messages for each respective group are output in the Web page at the specified point:

```
<HTML>
<BODY>
<FORM ACTION="custnum.w" METHOD="post">
<P>
<!--WSMSG NAME="CustNum" VALUE="Customer Number Errors" -->
<HR>
Enter Customer Number:
<INPUT TYPE="text" NAME="Customer_Number"> <BR>
<INPUT TYPE="submit" NAME="SUBMIT"> <BR>
<HR>
<!--WSMSG NAME="CustName" VALUE="Customer Name Errors" -->
Customer Name:
<INPUT TYPE="text" NAME="Customer_Name"> <BR>
Customer Phone:
<INPUT TYPE="text" NAME="Customer_Phone">
</P>
</FORM>
</BODY>
</HTML>
```

The tagmap utility procedure *install-path*/src/web/support/webmsg.p contains the default web.output control handler to output the HTML for this custom tag.

5.4.2 Passing Information Between Web Requests

There are four ways to pass information between Web requests:

- Cookies
- URL query strings
- Hidden form fields
- User fields

The first three techniques are available to all Web objects, stateless or state aware. However, user fields are only available for a single Web request or a WebSpeed transaction.

WebSpeed supports all of these techniques with API functions or method procedures. Some functions are specific to one technique and some combine these techniques to make information available to a Web object.

Cookies

Cookies allow you to pass information as part of the Web page HTTP header and are one of the commonly used techniques for identifying users of a Web site. In fact, WebSpeed transactions actually depend on a form of cookie passing supported by the WebSpeed Transaction Agent control program (`web-disp.p`) and the Broker.

Table 5–3 lists the API functions that support cookie passing.

Table 5–3: Cookie-passing API Functions

Function	Return Type	Description
delete-cookie	CHARACTER (Unknown, ?)	Deletes the cookie specified by its name, URL path, and Internet domain.
get-cookie	CHARACTER	Given a cookie name, returns one or more matching values delimited by the value of the WebSpeed global variable, SelDelim (comma, by default). If the cookie name is the unknown value (?), returns a list of all the cookie names.
set-cookie	CHARACTER	Outputs an HTTP Set-Cookie header with specified options, and returns the specified cookie value.

The definitions for these functions reside in *install-path*`/src/web/method/cookies.i`

You must invoke the delete-cookie and set-cookie functions within the output-header method procedure of a Web object. In embedded SpeedScript Web objects, you must include the output-header procedure definition in a statement escape tag. For SpeedScript-generating and HTML-mapping Web objects, WebSpeed provides standard output-header procedure definitions that you can modify in the WebSpeed Editor.

URL Query Strings

URL query strings are used to pass a wide range of information, from CGI requests to explicit values attached to dynamic links. WebSpeed provides several API functions to facilitate the building and reading of URLs, listed in Table 5–4.

Table 5–4: URL API Functions

Function	Return Type	Description
url-decode	CHARACTER	Decodes a URL form input from either CGI POST and GET request methods or encoded Cookie values, and returns the decoded string.
url-encode	CHARACTER	Encodes unsafe characters in a URL per RFC 1738 section 2.2, available at: `URL:http://ds.internic.net/rfc/rfc1738.txt`, plus ASCII values between 0 and 31 and between 127 and 255. Options modify the encode to handle URL query strings, persistent cookies, or a specified string of characters. Returns the encoded string.
url-field-list	CHARACTER	Encodes a list of name/value pairs from a list of names whose values are retrievable from the current request by the get-value API function. Parameters include the name list and delimiter. Returns the encoded list of name/value pairs.
url-field	CHARACTER	Encodes name/value pairs for use as an argument "field" to a URL. Parameters include the name, value, and delimiter for the pair. Returns the encoded name/value pair.
url-format	CHARACTER	Formats a URL from a base URL, name list, and delimiter. (The name list is encoded using url-field-list.) Returns encoded URL.

The definitions for these functions reside in *install-path*`/src/web/method/cgiutils.i`.

For most applications, the url-format function is the main API function to construct URLs. This function calls all the others in the table. To construct URL query strings, you also must call the set-user-field function to assemble the name/value pairs for output with the query string.

For reading URL query strings, the easiest and most common function to use is get-value, which searches several sources for the value associated with a name. For more information, see the "General Information Exchange" section.

URL query strings are limited in terms of the amount of information per URL. For larger lists of data that might accumulate in applications such as Internet shopping carts, you might choose to use hidden form fields.

Hidden Form Fields

Hidden fields provide a virtually inexhaustible means of passing information between Web requests. Based on user input from a form, you can return field values as hidden fields for the next Web request, thus maintaining a running record of data from prior requests that you pass from Web object to Web object and from Agent to Agent.

Table 5–5 lists the API functions that support hidden field construction. These functions construct complete HTML definitions for hidden fields in a form.

Table 5–5: Hidden Field API Functions

Function	Return Type	Description
hidden-field	CHARACTER	Returns a string containing an HTML hidden form field with HTML special characters encoded. Parameters include the name and value of the field.
hidden-field-list	CHARACTER	Formats and returns a list of hidden fields delimited by newline characters. Parameters include a list of field names retrievable from the current request by the get-value API function.

See *install-path*`/src/web/method/cgiutils.i` for definitions for these functions.

For most applications, hidden-field-list is the main API function for constructing hidden fields, because you can define one or more hidden fields with it. To construct a list of hidden fields, use the set-user-field function to define the name/value pairs for each hidden field. Then call hidden-field-list to construct the hidden fields from the user field list.

For reading individual hidden fields, use the get-value API function. For more information, see the "General Information Exchange" section.

User Fields

WebSpeed provides a global list that allows Web objects to communicate on the same Agent. This is the user field list. As a global data structure, several Web objects running in a single Web request or in the same WebSpeed transaction can pass information in user fields without passing parameters. Two API functions support user fields, as shown in Table 5–6.

Table 5–6: User Field API Functions

Function	Return Type	Description
get-user-field	CHARACTER	Returns the associated value for a specified user field that was set with set-user-field. If the user field name is specified as the unknown value (?), the entire list of user fields is returned.
set-user-field	LOGICAL	Sets the associated value for a specified user field. Parameters include the name and value of the field. If the number of fields is less than or equal to 255, the function returns TRUE. Otherwise, it returns FALSE.

The definitions for these functions reside in *install-path*`/src/web/method/cgiutils.i`.

User fields are name/value settings maintained as lists in WebSpeed global character variables. As such, these fields are available, without need of declaration, to all Web objects running on the same Agent for the same Web request or WebSpeed transaction (if the Agent is locked). Thus, where there is no WebSpeed transaction, user fields provide a means to pass data among several Web objects servicing a single request.

The set-user-field function has wide application in WebSpeed. It sets up name/value pairs for several other data passing API functions, including url-format, url-field-list (for query strings), hidden-field-list, and get-value.

5.4.3 General Information Exchange

WebSpeed supports several generally useful API functions for information exchange, some of which are most commonly used to handle Web requests. Table 5–7 lists these functions.

Table 5–7: General Information Exchange API Functions

Function	Return Type	Description
get-cgi	CHARACTER	Returns the value of a specified CGI variable, or returns the list of all CGI variables if the specified variable name has the unknown value (?). Returns blank (" “) if the name is invalid.
get-field	CHARACTER	Returns the associated value for a specified form field or query string, or returns the list of all form fields in the current request if the specified field name has the unknown value (?). Returns blank (" “) if the name is invalid.
get-value	CHARACTER	Returns the first available value associated with the name of a user field, a form field, a query string, or a cookie. If the specified name has the unknown value (?), returns the list of all user fields, form fields, and cookies. Returns blank (" “) if the name is invalid. This is the most commonly used API function.
html-encode	CHARACTER	Converts various ASCII characters in a string to their HTML representation. Returns the HTML-encoded string. This function is useful for constructing HTML output from SpeedScript program data. However, do not execute it more than once for a given string to avoid corrupting the encoding.

See *install-path*`/src/web/method/cgiutils.i` for definitions of these functions.

5.4.4 Managing Date and Time Information

The functions listed in Table 5–8 convert dates and times between local and UTC time, and format dates and times for different types of Web output.

Table 5–8: Date and Time API Functions

Function	Return Type	Description
convert-datetime	CHARACTER (Blank, " “)	Inputs a conversion option, a date specified with the SpeedScript DATE data type, and a time specified as the number of seconds since midnight (see the SpeedScript TIME function). Outputs the date and time converted from local time to UTC, from UTC to local time, or normalized to have a legal number of seconds in a day. (Conversion normalizes the seconds for output by default.)
format-datetime	CHARACTER	Formats and returns a date and time for Web use. Supported formats include "COOKIE" and "HTTP". The "COOKIE" format is useful for setting cookie expiration dates. The "HTTP" format is useful for dates in HTTP headers. Options also convert the date and time from local to UTC and normalize the result before formatting.

The definitions for these functions reside in *install-path*`/src/web/method/cgiutils.i`.

5.4.5 Generating Web Page Headers

The functions listed in Table 5–9 output Web page header information. For output-http-header to have any effect, you must execute it before output-content-type. Usually this is in the output-header procedure of a Web object.

Table 5–9: Web Page Header API Functions

Function	Return Type	Description
output-content-type	LOGICAL	Sets and outputs the MIME Content-Type header followed by a blank line. If the header is already output, no action is taken. If the specified content type is blank (" "), no Content-Type header is output, but other headers such as Cookies are output followed by a blank line. Returns TRUE if Content-Type header is output; otherwise, FALSE.
output-http-header	CHARACTER (Unknown, ?)	Outputs the specified HTTP header and associated value, followed by a carriage return (CR) and linefeed (LF). If the header is blank, CR and LF are still output.

See *install-path*/src/web/method/cgiutils.i for definitions of these functions.

5.4.6 Checking Configuration Options

Table Table 5–10 lists functions that you can use to verify WebSpeed configuration options. These functions allow you to alter application behavior based on the type of environment the WebSpeed Agent is running in.

Table 5–10: Configuration API Functions

Function	Return Type	Description
check-agent-mode	LOGICAL	Returns TRUE if the WebSpeed Agent is running in "Development", "Production", or "Evaluation" mode. Otherwise, returns FALSE.
get-config	CHARACTER	Returns the value of the specified WebSpeed Transaction Server configuration option.

See *install-path*`/src/web/objects/web-util.p` for definitions of these functions.

5.5 XML Support in SpeedScript

SpeedScript supports extensions that allow the use of XML through the Document Object Model (DOM) interface. These extensions provide the basic input, output, and low-level data manipulation capabilities required to use data contained in XML documents.

For more information about XML support, see *Progress External Program Interfaces*, which describes XML support in the context of Progress 4GL. However, the information also applies to SpeedScript, which is based on Progress 4GL.

5.6 Overview of Embedded SpeedScript

Embedded SpeedScript provides a way for you to build Web objects directly from standard HTML files by including SpeedScript as a scripting language.

An Embedded SpeedScript file looks like a static HTML file in which you embed a section of SpeedScript code using statement escapes. The following shows a simple example where Embedded SpeedScript is used to query the Sports2000 database:

escript1.htm

```
<HTML>
<HEAD>
<TITLE>My First Embedded SpeedScript File</TITLE>
</HEAD>
<BODY>
<H1>My First Embedded SpeedScript File</H1>

<SCRIPT LANGUAGE="SpeedScript">
FOR EACH Customer WHERE Name BEGINS "s":
   DISPLAY {&WEBSTREAM} Customer.
END.
</SCRIPT>
<HR>
</BODY>
</HTML>
```

The HTML code in boldface shows the embedded SpeedScript. Note that SpeedScript, like JavaScript, can be embedded by using the HTML <SCRIPT> tag. However, SpeedScript differs from JavaScript with respect to execution. JavaScript typically executes on the client. SpeedScript executes entirely on the WebSpeed Transaction Server. Server-side execution allows the embedded SpeedScript to reference Web object variables and database fields from anywhere in the HTML file.

The Web object generated from escript1.htm outputs a Web page that lists all customer records in the sample Sports2000 database with names that begin with the letter "s."

NOTE: {&WEBSTREAM} ensures that all DISPLAY statement output goes to the same stream as {&OUT}. For more information, see Chapter 6, "Handling DISPLAY Output."

When you write an HTML file that includes embedded SpeedScript, the AppBuilder converts the HTML file to a SpeedScript Web object that generates the actual Web page. This Web page can be a simple static or a complex dynamic Web page with programmatically varied content.

You can actually write embedded SpeedScript to generate one of two types of WebSpeed output file:

- Complete Web objects, ready to execute, that can handle both output to and input from the Web
- SpeedScript include files that you can include in the SpeedScript source code of other Web objects for later compilation

You can specify the type of output file to generate by using a <META> tag in the HTML head section. The default output file is a Web object. Embedded SpeedScript also supports <META> tags to specify HTTP options such as the character set of the HTML file. You can also specify <META> tag information in the HTML body using WebSpeed <!--WSMETA --> custom tags. For more information, see the "Using <META> and <!--WSMETA --> Tags" section.

5.7 Authoring Embedded SpeedScript Files

You can author an embedded SpeedScript file in most any Web authoring tool or in the Procedure Window in the AppBuilder. (You can start the Procedure Window from the AppBuilder main window by selecting Tools→ New Procedure Window.)

Most authoring tools accept some or all of the WebSpeed options for embedding SpeedScript. While an authoring tool might not accept some embedded SpeedScript options, there are usually alternative options available to accomplish the same end. Pure text editors like vi and the Procedure Window accept all of the embedded SpeedScript options.

Embedded SpeedScript options consist of a set of tags, some of which are created for WebSpeed and some that exist in other products. To minimize the chance of conflict, avoid using tags that are used by other server-side environments unless you have no other alternative. For more information, see the "How to Embed SpeedScript in HTML" section.

To create an embedded SpeedScript file using the Procedure Window in the AppBuilder, click on File→ New. Then, select Blank from the list of Web objects in the New dialog box. The Procedure Window opens with an HTML template containing basic syntax to enter embedded SpeedScript options.

5.8 How to Embed SpeedScript in HTML

WebSpeed provides two basic types of options to embed SpeedScript in an HTML file:

- Statement escapes
- Expression escapes

Statement escapes allow you to include one or more complete SpeedScript statements in the HTML, while expression escapes allow you to include SpeedScript expressions that have character formats (most all). You can code each type of escape using several different sets of matching start tags and end tags. This choice of tags minimizes the chance that your authoring tool will not accept the escape.

5.8.1 Statement Escapes

Table 5–11 shows the supported statement escapes.

Table 5–11: Embedded SpeedScript Statement Escapes *(1 of 2)*

Start Tag	Matching End Tag	Comments
<SCRIPT LANGUAGE = "SpeedScript">	</SCRIPT>	This escape is supported by most authoring tools.
<?WS>	</?WS>	These are WebSpeed-defined tags. The <? sequence is an SGML directive, but some authoring tools might not support it.
<SERVER>	</SERVER>	Navigator Gold supports these tags for server-side JavaScript. WebSpeed supports these tags because, although Navigator Gold also supports a <SCRIPT> tag, this tag does not allow you to specify the language.

Table 5–11: Embedded SpeedScript Statement Escapes *(2 of 2)*

Start Tag	Matching End Tag	Comments
<%	%>	These are Microsoft Active Server Pages (ASP) command tags.
<!--WSS	-->	This escape is a WebSpeed-defined HTML comment and is therefore supported by virtually all authoring tools. You might be able to use this escape to place embedded SpeedScript where some authoring tools consider other tags illegal, such as before <HTML> or after </HTML>.

A statement escape can enclose any number of complete SpeedScript statements. The embedded SpeedScript file in `escript2.htm` generates the same Web page as the example in `escript1.htm`, but it uses a statement escape instead of the `<SCRIPT>` tag:

escript2.htm

```
<HTML>
<HEAD>
<TITLE>My First Embedded SpeedScript File</TITLE>
</HEAD>
<BODY>
<H1>My First Embedded SpeedScript File</H1>

<?WS>
FOR EACH Customer WHERE Name BEGINS "s":
   DISPLAY {&WEBSTREAM} Customer.
END.
</?WS>
<HR>
</BODY>
</HTML>
```

5.8.2 Expression Escapes

Table 5–12 shows the supported expression escapes.

Table 5–12: Embedded SpeedScript Expression Escapes

Start Tag	Matching End Tag	Comments
‘	‘	The back tic (‘) is supported by Netscape LiveWire and is otherwise acceptable to most other authoring tools. This escape can be difficult to spot in code, depending on the font.
{=	=}	This escape is easier to spot than the back tic and is also acceptable to most authoring tools.
<%=	%>	These are Microsoft Active Server Pages (ASP) command tags. These tags might cause problems for some authoring tools because they use the angle brackets (< >).
<!--WSE	-->	This escape is a WebSpeed-defined HTML comment and is therefore supported by virtually all authoring tools. You might be able to use this escape to place embedded SpeedScript where some authoring tools consider other tags illegal, such as before <HTML> or after </HTML>.

An expression escape can enclose any sequence of valid SpeedScript expressions with a character output format, including variables, database fields, functions, and literal character strings. You can place expression escapes either alone or as part of an HTML string.

In `escript3.htm`, each SpeedScript expression is a database field inserted in an HTML table cell. Note how the statement escapes are distributed among the table tags to return a new customer record for each table row:

escript3.htm

```
<HTML>
<HEAD><TITLE>My Second Embedded SpeedScript File</TITLE></HEAD>
<BODY>
<H1>My Second Embedded SpeedScript File</H1>
<CENTER><TABLE BORDER="2">
<!--WSS FOR EACH Customer WHERE Name BEGINS "s": -->
<TR><TD> ‘Cust-num‘ </TD><TD> ‘Name‘ </TD><TD> ‘Phone‘ </TD></TR>
<!--WSS END. -->
</TABLE></CENTER>
</BODY>
</HTML>
```

5.9 Building Interactive Web Pages with Embedded SpeedScript

You can use embedded SpeedScript to build request-driven Web objects or Web objects that are called directly by other Web objects. Web objects built with embedded SpeedScript can use all of the state passing techniques available to any Web object. You can also build almost any type of dynamic Web page with embedded SpeedScript that includes forms as well as tables.

For some examples, see the "Embedded SpeedScript Examples" section in Chapter 4, "Sample Web Objects."

5.9.1 Passing Parameters

In `escript4.htm`, the embedded SpeedScript accepts a run-time parameter:

escript4.htm

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="SpeedScript">
/* Pass in the filter. */
DEFINE INPUT PARAMETER p_filter AS CHARACTER NO-UNDO.
</SCRIPT>
<TITLE>My Third Embedded SpeedScript File</TITLE>
</HEAD>
<BODY>
<H1>My Third Embedded SpeedScript File</H1>
<CENTER>
<H2>Table for name matching 'p_filter'</H2>
<TABLE BORDER="2">
<!--WSS FOR EACH Customer WHERE Name BEGINS p_filter: -->
<TR><TD> 'Cust-num' </TD><TD> 'Name' </TD><TD> 'Phone' </TD></TR>
<!--WSS END. -->
</TABLE>
</CENTER>
</BODY>
</HTML>
```

You call the Web object generated for this embedded SpeedScript file from another Web object or procedure, passing the parameter value in a RUN statement:

```
RUN escript4.r ("Johan").
```

5.9.2 Managing Dynamic Pages with Forms

You can use embedded SpeedScript to manage GET and POST requests for forms. The `browse.html` template provided in *install-path*`/src/web2/template` shows how you can do this. By default, this template generates a Web object that browses the Customer table of the Sports2000 database, using SalesRep as a query filter.

You can also generate a copy of `browse.html` by choosing File→ New→ Report Template in the AppBuilder.

Defining Local Data and Getting Input

The following sections describe several sections of this template. A `<SCRIPT>` tag introduces the first SpeedScript references in the HTML body (edited for brevity):

browse.html

```
<HTML>
   .
   .
   .
<SCRIPT language="SpeedScript">

  /* Modify the "&SCOPE-DEFINE..." lines to customize this browse. */

  &SCOPED-DEFINE Query-Table    Customer
  &SCOPED-DEFINE Query-Field    Name
  &SCOPED-DEFINE Query-Index    Name
  &SCOPED-DEFINE Filter-Field   Salesrep
  &SCOPED-DEFINE Display-Fields "<TR><TD>" Name "</TD><TD>"
    Cust-num "</TD></TR>"
  &SCOPED-DEFINE Result-Rows    10  DEF VAR jumpto-value AS CHAR NO-UNDO.
  DEF VAR {&Filter-Field}-key AS CHAR NO-UNDO.
  DEF VAR repo-to AS INTEGER NO-UNDO.
  DEF VAR i-count AS INTEGER NO-UNDO.
  DEF VAR JumpForm AS CHAR NO-UNDO.
  DEF VAR TmpUrl AS CHAR NO-UNDO.
  DEF VAR DelimiterField AS CHAR NO-UNDO.

  /* Get all the fields from the input */
  ASSIGN
    JumpForm = get-field("NoJumpForm":U)
    jumpto-value = get-field("jumpto-name":U)
    repo-to = INTEGER(get-field("repo-to":U)) NO-ERROR.
    {&Filter-Field}-key = get-field("{&Filter-Field}":U).

  /* Display "Jump To" Form.  This form will not be displayed if
     "NoJumpForm=yes" requested */
  IF JumpForm eq "":U THEN DO:
</SCRIPT>
```

Here, the file defines the local Web object environment, including customizable preprocessor definitions. Several calls to the get-field API function return user input for each request. Of particular interest is the get-field("NoJumpForm":U) call that returns a URL query string value. This initial script section terminates by testing this value to determine whether to generate a form to output on the next request. Note that the SpeedScript terminates with a DO statement header, thus beginning a SpeedScript DO block.

Defining Forms

Next comes an HTML form section that includes embedded SpeedScript expression escapes. This "Jump to" form allows the user to specify a starting Customer Name other than the default (the first) in alphabetical order:

browse.html

```
<FORM ACTION="'SelfURL'" METHOD="POST" NAME="JumpForm">
    <INPUT TYPE="hidden" NAME="repo-to" VALUE="-1000"><INPUT
    TYPE="hidden" NAME="sales-rep"
    VALUE="'{&amp;Filter-Field}-key'"><P><!--webbot
    BOT="HTMLMarkup" STARTSPAN --><CENTER><!--webbot BOT="HTMLMarkup"
    ENDSPAN --><STRONG>Jump to: <INPUT TYPE="text" SIZE="20"
    NAME="jumpto-name" VALUE="'jumpto-value'"><INPUT
    TYPE="submit" NAME="submit-jumpto" VALUE="Submit"> <!--webbot
    BOT="HTMLMarkup" STARTSPAN --></CENTER><!--webbot BOT="HTMLMarkup"
    ENDSPAN --></STRONG></P>
    <HR>
</FORM>
```

NOTE: The <!--webbot ... --> comments are added to work around certain defaults enforced by Microsoft FrontPage. For more information, see the Microsoft documentation on FrontPage.

SelfURL is a WebSpeed global variable provided in *install-path*/src/web/method/cgidefs.i that contains the URL of this Web object, relative to the server root directory. {&Filter-Field}-key is a local Web object variable (defined earlier) that specifies the filter for a database query. The embedded SpeedScript preprocessor changes '&' to '&' so a legal SpeedScript preprocessor reference ({&Filter-Field}) appears in the Web object source code. Finally, jumpto-value is the most recent Customer Name specified by the user to start the query.

Note that the embedded SpeedScript preprocessor turns this entire HTML form section into SpeedScript statements that appear in the DO block started earlier.

Managing SpeedScript Blocks and Conditional Execution

The next section is a small script section that begins by closing this DO block:

browse.html

```
<SCRIPT language="SpeedScript">
  END. /* Close of "IF JumpForm eq "":U THEN DO:" */

  /* Display "Results List" if "Jump To" Form posted or
     "NoJumpForm=yes" requested */
  IF repo-to ne 0 OR JumpForm ne "":U THEN DO:
</SCRIPT>

<P ALIGN="center"><STRONG>Results List:</STRONG></P>
<DIV ALIGN="center"><CENTER>

<TABLE BORDER="2">
```

This script section ends by starting another DO block if the Web object is either handling a POST from the form or running without generating the form. Note that the "Jump to" form includes hidden fields, including repo-to with a nonzero value to indicate that the form was displayed and the user is submitting it to return a query. This DO block begins with HTML preprocessed into SpeedScript to start a centered HTML table that handles the query output 10 items (Result-Rows) at a time.

The next script section includes the actual query code included in the DO block to resolve, reposition, and output the query rows, based on user input. To simplify embedded SpeedScript interaction between SpeedScript and HTML, this SpeedScript code also includes the necessary HTML to specify each row of the HTML table for the query. The table row is specified by the preprocessor reference {&Display-Fields}, which is defined in the first script section of the embedded SpeedScript file. (The actual query resolution and output is beyond the scope of this description. For more information, see the *install-path*/src/web/browse.html template.)

Following the query resolution and output section is another mixture of HTML and embedded SpeedScript code that concludes the query DO block:

browse.html

```
</TABLE>
</CENTER></DIV><SCRIPT LANGUAGE="SpeedScript">
    /* If Not the End of the Query, Increment Reposition pointer
       for Navigation.  If End of Query, force Query to start from
       beginning */
    IF AVAILABLE {&Query-Table}
      THEN repo-to = CURRENT-RESULT-ROW("Browse-Qry":U) + 1.
      ELSE repo-to = -1.

  END. /* Close of "IF repo-to ne 0 OR JumpForm ne "":U THEN DO:" */

  /* Display Navigation Links unless we are waiting for the very
     first "Jump To" to be submited */
  IF repo-to ne 0 THEN DO:
    TmpUrl = url-format(?,'jumpto-name,{&Filter-field},NoJumpForm',?).
    IF INDEX(TmpUrl,"?") > 0
      THEN DelimiterField = ?.
      ELSE DelimiterField = "?".
</SCRIPT>
```

This section also starts another DO block that begins by setting up the initial contents of the URL query string for any following request driven by a navigation panel (yet to be output).

Building URL Query Strings and Closing Out the Page

The final section completes the last DO block by sending out the navigation panel at the end of the Web page:

browse.html

```
<DIV
ALIGN="center"><CENTER>

<TABLE BORDER="2">
  <TR>
      <TD><A
      HREF="'TmpUrl%20+%20url-field('repo-to','1',DelimiterField)'"
      onmouseover="window.status='First '{&amp;Result-Rows}'';return true"
      onmouseout="window.status='';return true"><IMG
      SRC="/webspeed/images/first-au.gif" BORDER="0" WIDTH="16"
      HEIGHT="16"></A></TD>
      <TD><A
      HREF="'TmpUrl%20+%20url-field('repo-to',string(repo-to%20-%20({&amp;
        result-rows}%20*%202)),DelimiterField)'"
      onmouseover="window.status='Prev '{&amp;Result-Rows}'';return true"
      onmouseout="window.status='';return true"><IMG
      SRC="/webspeed/images/prev-au.gif" BORDER="0" WIDTH="16"
      HEIGHT="16"></A></TD>
      <TD><A
      HREF="'TmpUrl%20+%20url-field('repo-to',string(repo-to),
        DelimiterField)'"
      onmouseover="window.status='Next '{&amp;Result-Rows}'';return true"
      onmouseout="window.status='';return true"><IMG
      SRC="/webspeed/images/next-au.gif" BORDER="0" WIDTH="16"
      HEIGHT="16"></A></TD>
      <TD><A
      HREF="'TmpUrl%20+%20url-field('repo-to','-999',DelimiterField)'"
      onmouseover="window.status='Last '{&amp;Result-Rows}'';return true"
      onmouseout="window.status='';return true"><IMG
      SRC="/webspeed/images/last-au.gif" BORDER="0" WIDTH="16"
      HEIGHT="16"></A></TD>
  </TR>
</TABLE>
</CENTER></DIV><SCRIPT LANGUAGE="SpeedScript">
  END. /* Close of "IF repo-to ne 0 THEN DO:" */
</SCRIPT>
</BODY>
</HTML>
```

The onmouseover and onmouseout references specify JavaScript event handlers.

The navigation panel is built from an HTML table that includes GIF buttons associated with anchors back to this Web object. Each anchor contains a URL query string that specifies how to reconstruct and navigate the most recent query. The query DO block described earlier interprets this information. Note the use of %20 to enter spaces into each embedded SpeedScript expression escape that specifies a URL query string. This is inserted by the authoring tools and specifies the space that is required around SpeedScript string (+) and arithmetic (-) operators used to construct the query string. The embedded SpeedScript preprocessor replaces these with actual spaces in the Web object source code. Finally, a small script section terminates the navigation panel DO block followed by the final HTML tags.

From this description, you can see that an embedded SpeedScript file can generate practically any variety and size of Web page imaginable. In this sense, an embedded SpeedScript Web page can be seen to shrink and expand as it responds to user input and generates database output. With the basic input/output logic written, you can then make any adjustments to account for the server-side realities of Web object execution.

5.10 Using <META> and <!--WSMETA --> Tags

WebSpeed supports <META> tags and <!--WSMETA--> custom tags to specify how an embedded SpeedScript file is to be preprocessed. These tags provide two types of information:

- Whether (and how) a SpeedScript include file or a procedure file (Web object) is generated from the embedded SpeedScript file
- How the embedded SpeedScript Web object outputs HTTP header information

The following sections describe how to specify this information using <META> tags. However, some authoring tools allow you to specify <META> tags only in the header section (starting with the <HEAD> tag) of a Web page. If you are writing an HTML file fragment that is part of a Web page body, there is no place to include <META> tags, especially for file type information. WebSpeed provides the <!--WSMETA--> custom tag to work around this.

To use this WebSpeed tag in an embedded SpeedScript file, include the same options in the <!--WSMETA--> custom tag that you ordinarily include in the <META> tag. The <!--WSMETA--> custom tags are preprocessed exactly like <META> tags.

In the following sections, you can replace <META> tag references with <!--WSMETA--> tag references unless notified otherwise.

5.10.1 Specifying File Type Options

This is the <META> tag syntax for specifying the embedded SpeedScript file:

SYNTAX

```
< META NAME = "wsoptions" CONTENT =
"web-object-opt [ ,web-object-opt ] ... " | " get-options" >
```

SYNTAX: *web-object-opt*

```
include | no-compile | web-object | no-content-type | compile |
keep-meta-content-type
```

NAME = "wsoptions" indicates that the <META> tag specifies the file type to generate.

If CONTENT contains any combination of *web-object-opt*, the preprocessed embedded SpeedScript file is formatted as a complete SpeedScript Web object.

If CONTENT contains "include", the preprocessed embedded SpeedScript file is preprocessed as a SpeedScript include file. Thus, the output contains SpeedScript source code without any WebSpeed include file references (such as to `cgidefs.i`). The source does not include any references to the WebSpeed OutputContentType( ) procedure or any other code to generate an HTTP header. In all other respects, the file is ready to be referenced as an include file in the source code of some other Web object.

If CONTENT contains "no-compile", the preprocessed embedded SpeedScript file will not be compiled to r-code.

If CONTENT contains only "web-object", the preprocessed embedded SpeedScript file is formatted as a SpeedScript procedure with all the additional SpeedScript elements to make a Web object. This includes appropriate references to WebSpeed include files and the code to generate HTTP header information.

If CONTENT contains "no-content-type", the preprocessed embedded SpeedScript file is formatted as a SpeedScript procedure with all the additional SpeedScript elements to make a Web object. However, the Web object does not execute the code that generates HTTP header information at run time. This allows you to create an embedded SpeedScript Web object that you can call from another Web object that manages the Web page output.

If CONTENT contains "keep-meta-content-type", the include file or Web object is formatted to retain any HTTP header information (content type <META> tags) that is embedded in the HTML file at compile time. Some authoring tools add this tag automatically, whether or not you specify it yourself. In any case, each content type <META> tag can cause some browsers to restart document loading for each generated HTTP header it encounters. To prevent the browser disruption that this can cause, the AppBuilder omits any content type <META> tag from the generated file when you do **not** specify the keep-meta-content-type option. All content type <META> tags are commented out like this:

```
<!-- E4GL Disabled: META HTTP-EQUIV="Content-Type" ... -->
```

By default, the AppBuilder prevents any conflict between additional HTTP header information embedded in the HTML file and the HTTP header information generated at run time. For more information on using content type <META> tags in your embedded SpeedScript, see the "Specifying HTTP Header Information" section.

NOTE: During preprocessing of the embedded SpeedScript file, the header information in all content type <META> tags is retrieved for the Web object to generate at run time using the output-content-type function.

Any unknown options that you specify in the CONTENT value are ignored.

The "get-options" option is for development applications that manage embedded SpeedScript files, such as to automate a build process. Generally, you pass this option to the embedded SpeedScript preprocessor (*install-path*`/tty/webutil/e4gl-gen.r`) to return information about another HTML file. The preprocessor takes three parameters, in this order:

1. Pathname of the HTML file to preprocess (INPUT, AS CHARACTER)

2. Pathname of the output Web object or include file to generate (INPUT-OUTPUT, AS CHARACTER)

3. Comma-delimited list of file type options (INPUT-OUTPUT, AS CHARACTER)

You pass "get-options" as the third parameter and the preprocessor returns a list of all the file type options specified in <META NAME="wsoptions"...> and <!--WSMETA NAME="wsoptions"...--> tags. If any of these tags are found, the string "wsoptions-found" is also appended to the list. You can use this information to determine whether a file contains embedded SpeedScript or is only a static HTML file. You can also add your own CONTENT options to these tags, and although the preprocessor ignores them for its own use, it does return them in the list.

To generate a Web object that omits the HTTP header from its output, you might begin your embedded SpeedScript file like this:

meta.html

```
<HTML>
<HEAD>
<META NAME="wsoptions" CONTENT="web-object, no-content-type">
</HEAD>
<BODY>
   .
   .
   .
```

NOTE: Depending on options you choose in AppBuilder, you can generate an include or procedure file whether or not you use this <META> tag in the embedded SpeedScript file. If you add this tag to the embedded SpeedScript file, the embedded SpeedScript preprocessor combines the options you specify with any provided by AppBuilder.

If you write a file management procedure and call the preprocessor for `meta.html`, passing "get-options":

```
DEFINE VARIABLE speedfile AS CHARACTER.
DEFINE VARIABLE wsoptions AS CHARACTER.
ASSIGN speedfile = "meta.w"
        wsoptions = "get-options".
RUN tty/webutil/e4gl-gen.r
  (INPUT "meta.html",
   INPUT-OUTPUT speedfile,
   INPUT-OUTPUT wsoptions).
```

the preprocessor returns "web-object, no-content-type, wsoptions-found" as the value of wsoptions.

5.10.2 Specifying HTTP Header Information

Content type <META> tags allow HTML files to specify HTTP header information with the following syntax:

SYNTAX

```
< META HTTP-EQUIV = "Content-Type" CONTENT = "http-info" >
```

For this tag, "*http-info*" can be any valid string of HTTP header information, as in this example that specifies a character set for the HTML file:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-2">
</HEAD>
<BODY>
   .
   .
   .
```

You can also include this tag in embedded SpeedScript files that generate Web objects. If included, the preprocessor generates code for the Web object that outputs the specified HTTP header information at run time (using the output-content-type function). The default HTTP header string generated for Web objects is "text/html".

NOTE: By default, embedded SpeedScript Web objects output the preprocessed header information at runt time but suppress output of all content type <META> tags in the Web page. To retain these <META> tags in the HTML at run time, specify "keep-meta-content-type" in a file type options <META> tag. For more information, see the "Specifying File Type Options" section.

For embedded SpeedScript files, this tag is especially useful when the content is created using an alternative character set. However, in addition to specifying the character set in this tag, you must ensure that the Stream Code Page (-cpstream) value for the Transaction Server specifies the same character set. Otherwise, the Web object might generate a Web page with unpredictable results.

5.10.3 Generating Information Prior to HTTP Header Output

For an embedded SpeedScript Web object, you can generate additional header information (such as cookies) or perform any other processing or authentication before the Web object outputs the specified HTTP header information. If you define an internal procedure named output-headers in your embedded SpeedScript file, the resulting Web object executes this procedure before generating the HTTP header. If output-headers returns an error (ERROR-STATUS:ERROR = TRUE), your embedded SpeedScript Web object returns without generating output to the Web. The WebSpeed include file *install-path*/src/web/method/e4gl.i contains the logic for handling this procedure.

To code an embedded SpeedScript output-header procedure with the Procedure Window in the AppBuilder, you must include the procedure definition inside a statement escape. The Procedure Window has no way to insert an output-header procedure template into embedded SpeedScript.

5.11 Developing Embedded SpeedScript Web Objects

Whatever authoring tool you use to create an embedded SpeedScript file, you must use the AppBuilder to generate the resulting Web object or include file. The AppBuilder allows you to create the embedded SpeedScript file and generate its target WebSpeed file, or you can author the embedded SpeedScript file using another tool and load it into the AppBuilder later.

5.11.1 Building Embedded SpeedScript Web Objects

The AppBuilder provides an option to create a new Web object by loading an existing HTML file. If the HTML file is, in fact, an embedded SpeedScript file (contains any statement or expression tags), AppBuilder then loads it for you to generate a SpeedScript include file or procedure file directly from it. If the HTML file is not an embedded SpeedScript file, but a standard static HTML file, the AppBuilder creates a separate HTML-mapping Web object for the loaded HTML file.

Unlike HTML-mapping Web objects, the AppBuilder always compiles and generates the r-code file for an embedded SpeedScript Web object. This is because all of the HTML for the generated Web page is contained in the Web object. Thus, the embedded SpeedScript file remains the entire source for the Web object it generates. Because you can modify HTML-mapping Web objects separately from the HTML files that they use, the AppBuilder must save the SpeedScript source for HTML-mapping Web objects as a starting point for changes. Thus, you might add an include file generated from embedded SpeedScript source to an existing HTML-mapping Web object.

5.11.2 Running Embedded SpeedScript Web Objects

You can run an embedded SpeedScript Web object like any other Web object by requesting its URL in a browser. Typically, you specify the r-code filename of the Web object (with the .r extension) in the URL. If you choose to specify the embedded SpeedScript filename instead (with the .htm or .html extension), the servicing Transaction Agent tries to locate the corresponding r-code file to execute and returns an error if it cannot find it.

6

Handling DISPLAY Output

WebSpeed provides support for Web formatting in the SpeedScript DISPLAY (and SQL SELECT) statement. In effect, this support allows much of the automatic output formatting of the DISPLAY statement to be reflected in the Web page. You can also control how this formatting is mirrored in the HTML.

This chapter include the following information:

- Working with DISPLAY output
- Changing DISPLAY output defaults

6.1 Working with DISPLAY Output

The SpeedScript DISPLAY statement provides automatic output formatting suitable for a report or display terminal. WebSpeed handles this output in different ways, depending on how you use it. WebSpeed also provides mechanisms to change the way it handles this output.

When using the DISPLAY statement, you must decide two issues:

- Where the output is going
- How the output is going to look

6.1.1 Directing DISPLAY Output

Like all SpeedScript output statements, the DISPLAY statement always directs output to a particular stream. In WebSpeed, this output stream must also be directed to the "WEB" device (your Web server). WebSpeed provides two ways to direct DISPLAY output to a stream:

- Use the WebSpeed-defined SpeedScript output stream
- Define and use a new SpeedScript output stream

In general, it is better to use the WebSpeed-defined output stream to maintain well-managed output to a Web page. The standard WebSpeed PUT statement directive ({&OUT}) uses the WebSpeed-defined output stream. Each output stream has its own buffer management. Because of this, if you output to more than one stream for a single Web request, the actual sequence of output to the Web page can be unpredictable. It might be different than the sequence specified by the order of output statements in your Web object.

Using the WebSpeed-defined Output Stream

WebSpeed provides two SpeedScript preprocessor definitions to help you use the WebSpeed-defined output stream:

- &GLOBAL-DEFINE WEBSTREAM STREAM WebStream
- &GLOBAL-DEFINE DISPLAY DISPLAY {&WEBSTREAM}

These definitions appear in the *install-path*/src/web/method/cgidefs.i include file used by all Web objects. Thus, the two DISPLAY statements in the following example yield the same result:

```
FIND FIRST Customer.
DISPLAY {&WEBSTREAM} Name Phone.
{&DISPLAY} Name Phone.
```

Use one of the forms shown in boldface to guarantee that all of your DISPLAY-formatted output appears in the Web page at the corresponding point where the code appears in your Web object.

Defining and Using a New Output Stream

To define and use a new output stream, WebSpeed provides two statements:

- DEFINE STREAM statement—Defines a stream.
- OUTPUT TO statement with the "WEB" option—Opens and directs the specified stream to your Web server.

If you do not specify a stream in the OUTPUT TO statement, WebSpeed uses a default unnamed stream to send your output to the Web.

The following example defines a new stream and directs it to the Web:

```
FIND FIRST Customer.
DEFINE STREAM MyStream.
OUTPUT STREAM MyStream TO "WEB".
DISPLAY STREAM MyStream Name Phone.
```

This statement opens the default unnamed stream and directs it to the Web:

```
FIND FIRST Customer
OUTPUT TO "WEB".
DISPLAY Name Phone.
```

Both of these examples use the DISPLAY statement directly, sending its output to the Web. However, in the following example, there is no way to guarantee which will appear first in the Web page, the customer name or the customer address:

```
FIND FIRST Customer
OUTPUT TO "WEB".
DISPLAY Name Phone.
DISPLAY {&WEBSTREAM} Address City Postal-Code.
```

For more information on the statements used in these examples, see the *Progress Language Reference*.

6.1.2 Formatting DISPLAY Output

In general, DISPLAY output is formatted using SpeedScript frames. These are not HTML frames, but are logical structures maintained by SpeedScript to manage its output. There are several types of SpeedScript frames you can use to stream output to the Web, and this output can be organized into still larger, multi-frame stream pages. Stream pages are not Web pages, but logical sections of a Web page containing one or more SpeedScript frames. For information on the available types of SpeedScript frames, see the Frame Phrase entry in the *Progress Language Reference*.

Keep in mind that these SpeedScript frames and the larger multi-frame stream pages are structures that WebSpeed uses to format a single Web page for a single request. They do not, in and of themselves, return as successive individual Web pages, such as appears for large result sets from a Web search engine. They merely provide a means to partition and give structure to a single Web page using DISPLAY and other SpeedScript statements.

NOTE: Stream paging applies both to DISPLAY and PUT ({&OUT}) statement output. However, only DISPLAY (and SQL SELECT) statements output SpeedScript frames.

You can format DISPLAY statement output using two mechanisms:

- Paging options of the OUTPUT TO "WEB" statement to generate multi-frame stream pages.
- HTML attributes of the WEB-CONTEXT handle to specify HTML that supports or modifies the formatting in SpeedScript frames and pages.

Paging Options of the OUTPUT TO "WEB" Statement

The two paging options include:

- PAGED—Generates Web page output in 56-line stream pages.
- PAGE-SIZE *number-of-lines*—Generates Web page output in stream pages of a specified number of lines.

To open an unnamed output stream with 25-line stream pages, you specify this:

```
OUTPUT TO "WEB" PAGE-SIZE 25.
```

Each stream page contains the following output ordered and organized into one to three sections, all within the number of lines specified for the stream page:

1. Any content from activated SpeedScript PAGE-TOP frames for the current OUTPUT stream. Otherwise, the stream page begins with the second section.

2. Any other DISPLAY or PUT statement output for the current OUTPUT stream. If DISPLAY output is for a SpeedScript DOWN frame, any column headers are printed in the following occurrences:

   - At the start of an iteration block.
   - At the start of the stream page or immediately after any output from PAGE-TOP frames.

   If DISPLAY output is for a SpeedScript frame other than a DOWN frame, the entire content from the specified frame is output for each DISPLAY statement.

   All output in the second part of the stream page continues until the line number required to start any activated PAGE-BOTTOM frame or after the last line of the stream page, which ever comes first. If PAGE-BOTTOM frame output is activated, the stream page continues with the third section. Otherwise, the stream page is complete.

3. Any content from activated SpeedScript PAGE-BOTTOM frames for the current OUTPUT stream until the stream page is complete.

NOTE: Activate SpeedScript PAGE-TOP and PAGE-BOTTOM frames using DISPLAY statements. These statements do not immediately output the frames, but make them ready for output at the appropriate point in a stream page. Note also that PUT statements (most often coded using the WebSpeed {&OUT} preprocessor reference) can send output in the second section of a stream page. But this output is not part of any SpeedScript frames, which are used only by DISPLAY statements.

WebSpeed also provides a means to automatically separate successive stream pages in the HTML using a specified divider. The HTML for this divider is specified by the HTML-END-OF-PAGE attribute of the WEB-CONTEXT system handle. For more information, see the "HTML Attributes of the WEB-CONTEXT Handle" section.

For more information on setting these options, see the "Changing Output Format Defaults" section in this chapter.

HTML Attributes of the WEB-CONTEXT Handle

The WEB-CONTEXT handle provides a set of attributes to modify the formatted output in a Web page. Each attribute is a character string that contains HTML you want to add to a specified portion of SpeedScript frame and paged output. Any settings that you make to these attributes last only until the next Web request. Each request causes the Agent to reset these attributes to the default values. This is true for both stateless and state-persistent Web objects.

Table 6–1 describes how to use these attributes.

Table 6–1: WEB-CONTEXT Handle HTML Attributes *(1 of 2)*

Attribute (Location in Output)	Description
HTML-END-OF-LINE (End of Line)	Defaults to the newline character (ASCII 10; '~n'; '\n'). A null string value causes a NEWLINE character (not a null string) to be output. You might set this to . However, depending on the other attribute values, using the NEWLINE rather than the tag can result in more readable output when viewing document source in a browser. Output at the end of each data row for the current iteration of a DOWN frame.
HTML-FRAME-BEGIN (Before a SpeedScript frame)	Defaults to "<PRE>". Generally, if you change this value you must change the value of HTML-FRAME-END. Output only before the data rows for the current iteration of a DOWN frame, not to column headers (see also HTML-HEADER-BEGIN and HTML-HEADER-END). Applies to any side-labels displayed in the frame, whether or not the frame is a DOWN frame.
HTML-FRAME-END (After a SpeedScript frame)	Defaults to "</PRE>". Generally, if you change this value you must change the value of HTML-FRAME-BEGIN. Output at the end of the data rows for the current iteration of a DOWN frame.
HTML-HEADER-BEGIN (Before the column headers of a SpeedScript frame)	Defaults to "<PRE>". Generally, if you change this value you must change the value of HTML-HEADER-END. Output at the beginning of the column header section of a DOWN frame.

Table 6–1: WEB-CONTEXT Handle HTML Attributes *(2 of 2)*

Attribute (Location in Output)	Description
HTML-HEADER-END (After the column headers of a SpeedScript frame)	Defaults to "</PRE>". Generally, if you change this value you must change the value of HTML-HEADER-BEGIN. Output at the end of the column header section of a DOWN frame.
HTML-TITLE-BEGIN (Before a SpeedScript frame title)	Defaults to the null string (""), no text. Generally, if you change this value you must change the value of HTML-TITLE-END. Output before the frame's TITLE value. Setting to a color or bold tag might improve readability.
HTML-TITLE-END (After a SpeedScript frame title)	Defaults to the null string (""), no text. Generally, if you change this value you must change the value of HTML-TITLE-BEGIN. Output after the frame's TITLE value.
HTML-END-OF-PAGE (Between stream pages)	Defaults to "<HR>". Output between stream pages to visually break up the sectioning caused by the PAGED or PAGE-SIZE options of the OUTPUT TO "WEB" statement. Does not affect the line count of any stream page.

For information on how to set and read these attributes, see the WEB-CONTEXT Handle entry in the *Progress Language Reference*. For information on setting new defaults for these attributes, see the "Setting HTML Attribute Defaults" section.

6.2 Changing Output Format Defaults

You can change output format defaults for both the WebSpeed-defined output stream and the WEB-CONTEXT handle HTML attributes.

6.2.1 Setting {&WEBSTREAM} Defaults

By default, WebSpeed defines output to the Web ({&WEBSTREAM}) as a nonpaging stream. You can modify the WebSpeed definition at any point in a Web object using the OUTPUT CLOSE statement and OUTPUT TO "WEB" statement paging options. (See the "Formatting DISPLAY Output" section.) This example changes {&WEBSTREAM} several times in the Web object, first to 25-line, 10-line, and then no stream paging:

```
OUTPUT {&WEBSTREAM} CLOSE.
OUTPUT {&WEBSTREAM} TO "WEB":U PAGE-SIZE 25.
                .
                .
                .
OUTPUT {&WEBSTREAM} CLOSE.
OUTPUT {&WEBSTREAM} TO "WEB":U PAGE-SIZE 10.
                .
                .
                .
OUTPUT {&WEBSTREAM} CLOSE.
OUTPUT {&WEBSTREAM} TO "WEB":U.
```

NOTE: Stream paging applies both to DISPLAY and PUT ({&OUT}) statement output. However, only DISPLAY (and SQL SELECT) statements output SpeedScript frames.

6.2.2 Setting HTML Attribute Defaults

You can change the initialized defaults for all the HTML attributes of the WEB-CONTEXT handle. (See the "Formatting DISPLAY Output" section.) Create a text file in your PROPATH with the name `display.dat`. In this file, you can specify a new HTML definition for one or all of these attributes. Any definitions that appear in this file override the hard-coded WebSpeed defaults both at startup and during the initialization of each Web request service.

Entering Attribute Definitions

These are the basic rules for entering attribute definitions in `display.dat`:

- Any line beginning with pound (#) is a comment. No lines can precede or follow a definition except comment lines (no blank lines).
- The first character of the first line of a definition (if not pound (#)), is the *delimiter* for the definition.

This is the syntax for an HTML attribute definition:

SYNTAX

```
{delimiter}{HTML-attribute}{delimiter}[#]
[HTML-definition]{delimiter}
```

You must enter each of these elements in order with no intervening spaces, except as noted.

delimiter

The first character of the first line of each definition. You can specify a different character for each attribute definition. Any text following on the same line as the third occurrence of *delimiter* is ignored as comment until the end of the line. The next line must be a pound (#) comment line or start another attribute definition.

HTML-attribute

The name of the WEB-CONTEXT HTML attribute you are defining.

#

(Optional) If specified, pound (#) allows you to start *HTML-definition* on the next line (for readability only) without the preceding NEWLINE character becoming part of the definition.

HTML-definition

(Optional) Any HTML you choose for the definition, including spaces and NEWLINEs. The definition starts with the first character after the second (and preceding) *delimiter* unless you enter pound (#). If you enter pound (#), the definition starts with the first character after any immediately following NEWLINE. The definition continues for as many lines as you want, up to the third (and final) *delimiter*. If you enter no characters between the second and third *delimiter*, the definition is the null string. No checking is done to determine if the definition makes sense for the specified *HTML-attribute*.

Examples

The following text contains two associated HTML attribute definitions:

```
# This is an example of using the '#' after the second delimiter.
:HTML-FRAME-BEGIN:#
<FONT COLOR="BLUE">
<PRE>:
# Pay attention to the first character of the next definition. I am
# changing the delimiter that I am using.
+HTML-FRAME-END+#
</PRE>
</FONT>+
```

Note how the HTML-FRAME-BEGIN and HTML-FRAME-END reverse the order of the respective start and end tags to ensure their proper placement in the Web page.

This is a definition for a null string:

```
:HTML-END-OF-LINE::
```

NOTE: After creating or modifying `display.dat`, you must restart your WebSpeed Transaction Server for this file to take effect.

7

Generating HTML Visualizations

This chapter describes an implementation of HTML table generation installed with WebSpeed as a prototype for this visualization support, and describes:

- Using the support for HTML table visualizations
- Generating HTML tables with a custom tag
- Generating HTML tables Directly from SpeedScript
- Adding a customized file as an AppBuilder template
- Generating other visualizations

7.1 HTML Visualizations

WebSpeed provides support for dynamically generating HTML visualizations for a Web page. You can generate these visualizations in an HTML-mapping Web object using the custom tag, <!--WSTAG -->. This tag allows you to specify a set of HTML attributes that direct your Web object to perform any custom input or output that you design. By placing the tag in your HTML file, the Web object performs the specified output action when it scans the tag and returns any input for any HTML form element that you map with the tag. These actions are determined by a visualization procedure that you specify in the tag.

For non-HTML-mapping Web objects, you can execute the visualization procedure directly from SpeedScript or embedded SpeedScript to accomplish the same effects.

NOTE: For information on embedded SpeedScript files, see Chapter 5, "SpeedScript."

7.2 Using the Support for HTML Table Visualizations

To generate an HTML table using the installed visualization support:

- For HTML-mapping support, WebSpeed uses two tagmap utility procedures, *install-path*`/src/web/support/tagrun.p` and *install-path*`/src/web/support/tagparse.p` and relies on you to:
  - Include this custom tag where you want the table to appear in your mapped HTML file: `<!--WSTAG NAME="mytbl" FILE="table.html" -->`.
  - Customize the embedded SpeedScript file based on the installed template, *install-path*`/src/web/template/table.html`
- For SpeedScript-generating or embedded SpeedScript support, run your customized version of the visualization procedure template, *install-path*`/src/web/template/table.html`, in the Web object.

7.3 Generating HTML Tables with a Custom Tag

Using the <!--WSTAG --> custom tag installed with WebSpeed, you can include a dynamically generated table in the Web page generated by an HTML-mapping Web object. As with any custom tag you use for HTML-mapping, you must have the tag defined in your `tagmap.dat` file.

7.3.1 Rules for Using Custom Tags

The rules for creating the custom tag are similar to other standard HTML tags:

- The delimiter that separates HTML attribute settings is a space.
- Quote the attribute value if it has an embedded space. Otherwise, quotes are not necessary.

7.3.2 Using the <!--WSTAG--> Custom Tag

This is the default definition for the <!--WSTAG --> tag installed in `tagmap.dat`:

```
!--WSTAG,,,fill-in,web/support/tagrun.p
```

This is a good example of how you might define your own custom tags. At Web object generation time, the Workshop associates the <!--WSTAG --> custom tag with a SpeedScript fill-in field. That is, a local variable (fill-in) appears in the Web object after an HTML file (that contains <!--WSTAG -->) has been opened and saved as a new Web object.

As with the other utility procedures in WebSpeed, at run-time, `tagrun.p` is run in combination with a few other procedures to dynamically generate an HTML table.

To use the <!--WSTAG --> custom tag in an HTML file, you must specify two HTML attributes associated with the tag in order to generate a table or other custom HTML output:

- NAME
- FILE

NAME Attribute

As with other HTML fields, the NAME attribute's value is used as the associated WebSpeed object name and label in the Web object. In the case of <!--WSTAG -->, the associated WebSpeed object is a fill-in field.

FILE Attribute

The FILE attribute in <!--WSTAG --> provides the name of the SpeedScript procedure that generates the HTML. If FILE is set to a filename, `tagrun.p` runs the specified file as a SpeedScript procedure. If FILE is set to "screenvalue", `tagrun.p` retrieves the value of the SpeedScript SCREEN-VALUE attribute on the fill-in field associated with <!--WSTAG --> and uses this value as the name of the SpeedScript procedure to run.

These are examples of the three techniques, where `tagrun.p` runs the SpeedScript procedure `mytbl.p` and then runs the file specified by the value of the associated fill-in field:

```
<!--WSTAG NAME=mytbl FILE=mytbl.p -->
<!--WSTAG NAME=mytbl FILE=screenvalue -->
<!--WSTAG NAME=mytbl -->
```

With the third example, you must create a custom web.output control handler in Workshop and generate the table directly.

7.3.3 How tagrun.p Interacts with the Custom Tag

As with the other utility procedures in the *install-path*`/src/web/ support` directory (and referenced in the `tagmap.dat` file), the web.output trigger procedure for `tagrun.p` is run when the output-fields event procedure has been dispatched for a Web object.

Web.output takes hWid and fieldDef as input parameters; where hWid is the widget handle to the WebSpeed fill-in field associated with the custom tag, and fieldDef is the HTML field definition that appears in the HTML file. The output procedure then runs `tagparse.p` to parse the field definition for a specified tag attribute (in this case, NAME or FILE), both of which are passed as input parameters. After parsing fieldDef, `tagparse.p` returns the attribute value.

If the value of the FILE attribute is "screenvalue", the `tagrun.p` web.output trigger procedure retrieves the SCREEN-VALUE widget attribute of the fill-in field object associated with the tag and runs the specified procedure. This procedure then generates the HTML table.

If the value of the FILE attribute is a filename, then the `tagrun.p` web.output trigger procedure runs the specified procedure to generate an HTML table. If you do not specify a FILE attribute, then you must define a custom web.output control handler to generate the HTML table.

7.3.4 Using the Table Template

The `table.html` file provides an embedded SpeedScript template to build your table procedure. The installed template is written to output a two-column table of customer names and phone numbers from the sample Sports2000 database installed with WebSpeed.

Follow these steps to customize this template:

1 ♦ Make a copy of the table template.

2 ♦ Customize the template as needed.

3 ♦ Open the template in Workshop.

4 ♦ Compile the template to generate an r-code file.

5 ♦ Do one of the following:

a) Specify the generated r-code file (.r) in the HTML file using the FILE attribute of the custom tag.

b) Specify the generated r-code file (.r) as the initial value of the fill-in field object associated with the custom tag in the HTML-mapping Web object. (Skip Step 6.)

6 ♦ Open the HTML file from Step 5a in Workshop to generate an associated HTML-mapping Web object.

7 ♦ If you want to make the customized template available directly from AppBuilder, modify *`install-path`*`/src/template/web.cst` to install the HTML file in the AppBuilder New dialog box.

7.3.5 Compiling the Table Template

Because the table template is an embedded SpeedScript file, you must compile it to generate a `.r` file. You can compile your copy of the template in the Workshop Files or Tools view. Either view allows you to open the template file and compile it by clicking the Compile icon in the Workshop Main frame.

You specify this generated `.r` file in the custom tag in the HTML file. You can specify this file with the r-code extension (`.r`):

```
<!--WSTAG NAME="mytbl" FILE="mytbl.r" -->
```

7.4 Generating HTML Tables Directly from SpeedScript

You can generate an HTML table using embedded SpeedScript or a SpeedScript-generating Web object by directly calling the procedure based on your customized `table.html`. You do not need the custom tag because the embedded SpeedScript or SpeedScript-generating file contains all the HTML as well as your SpeedScript code.

There are several techniques for calling the table procedure. These techniques are similar to the custom tag method in that they require the same customization of the table template file and compilation of the file to generate a Web object (`.w`). However, they differ in how the generated Web object file is called. In the following descriptions of these techniques, `table.w` is the table-generating Web object:

- You can call `table.w` from another Web object (`mywo.w`) that was generated from an embedded SpeedScript HTML file. For example, `mywo.html` is an embedded SpeedScript file that contains a line: `RUN table.r.` You compile `mywo.html`, generating `mywo.r`, which you request from a browser.

- You can call `table.r` from a Web object (`mywo.w`) that is generated as an HTML-mapping or the CGI Wrapper Web object. In either case `mywo.w` calls `table.r` directly from the process-web-request procedure. In the HTML-mapping Web object, you might call `table.r` after dispatching the output-fields event procedure.

- You might request `table.r` directly from a browser. In this case, when customizing `table.html`, be sure to add the <HTML>, <HEAD>, and <BODY> tags.

7.5 Adding a Template to AppBuilder

The following shows the contents of *install-path*/src/template/web.cst, which defines the entries in the AppBuilder New dialog box:

web.cst

```
/* web.cst - WebSpeed 3.0 custom objects file */

*NEW-WEBOBJECT  HTML Mapping
NEW-TEMPLATE    src/web2/template/html-map.w

*NEW-WEBOBJECT  Report
NEW-TEMPLATE    src/web2/template/wreport.w

*NEW-WEBOBJECT  Detail
NEW-TEMPLATE    src/web2/template/wdetail.w

*NEW-WEBOBJECT  Blank
NEW-TEMPLATE    src/web/template/script.html

*NEW-WEBOBJECT  CGI Wrapper
NEW-TEMPLATE    src/web2/template/wrap-cgi.w

*NEW-WEBOBJECT  Frameset
NEW-TEMPLATE    src/web/template/frameset.html

*NEW-WEBOBJECT  Main
NEW-TEMPLATE    src/web/template/main.html

*NEW-WEBOBJECT  Report Template
NEW-TEMPLATE    src/web/template/browse.html

*NEW-WEBOBJECT  Table
NEW-TEMPLATE    src/web/template/table.html
```

If you created a new, customized template, you could add the template to your menu by adding a label and a pathname to the web.cst file.

NOTE: Generally you would use one of the templates referenced in web.cst as the basis for a new template. However, be aware that embedded SpeedScript templates require an additional .dat file. For example, wreport.w references a file named brwstmpl.dat. Within brwstmpl.dat there are specifications for input fields in the wizard. These specifications are surrounded by double pound signs (##). For more information on how to customize the AppBuilder, see the *Progress AppBuilder Developer's Guide*.

7.6 Support for Other HTML Visualizations

You can support other visualizations, such as Java controls, in the same manner as the HTML table described in the previous sections. As with HTML tables, you can make use of the <!--WSTAG --> custom tag or call the template Web object directly to output data using other visualizations. The key is the template file. For example, supporting a Java Grid Control requires creating a template similar to the `table.html` template. You have to output HTML in a format that is readable to the control. This format is generally specified in the control's API.

8

Controlling WebSpeed Transactions

A WebSpeed transaction maintains a context between a single Web browser and a single WebSpeed Agent. This chapter describes how you create and control Web objects within WebSpeed transactions. Topics include:

- Defining state
- Understanding WebSpeed transaction control
- Primary and secondary Web objects
- Implementing transaction control
- Transaction control with embedded SpeedScript Web objects
- Transaction control with CGI wrapper Web objects
- Transaction control with HTML-mapping Web objects
- Handling state-aware time-outs

8.1 Defining State

Web transactions are, by nature, *stateless*. In other words, there is no sustained connection between a Web server and a client. Also, the Web server does not maintain any information about a client for future reference. Each transaction terminates after a response (usually an HTML page) is returned to the browser.

However, it is also possible to implement WebSpeed applications that are *state-passing* or *state-persistent*. State-passing applications store information about prior transactions on the client side. State-persistent applications maintain a connection between the client and the WebSpeed Agent for some period of time.

8.1.1 Web Object States

The basic building blocks for all WebSpeed applications are Web objects. There are two states that can apply to Web objects:

- Stateless
- State-aware

Stateless Web objects are run and destroyed within a single web request. The WebSpeed Agent which services them retains no context for the next time it runs the object.

State-aware Web objects maintain context running persistently on a locked WebSpeed Agent. The locked Agent is dedicated to the requesting client browser and awaits a response only from that browser until a specified time-out period has passed.

NOTE: Of the three basic types of Web objects, you can only make CGI Wrapper and HTML-mapping Web objects state-aware. Embedded SpeedScript Web objects can run as stateless Web objects on a locked Agent, but cannot themselves lock an Agent. They lack the method procedures and structure required to interact with the Agent control program.

8.1.2 WebSpeed Application States

From stateless and state-aware Web objects, you can create three kinds of WebSpeed applications:

- Stateless
- State-passing
- State-persistent

Stateless WebSpeed application are composed entirely of stateless Web objects and static files. No information is retained between Web requests.

State-passing WebSpeed applications are also composed entirely of stateless Web objects and static files. However context information is passed to the WebSpeed Agent from the client through URL query strings, HTML form fields, and cookies.

State-persistent applications are composed of one or more state-aware Web objects. The WebSpeed Agent is locked to the browser for a specified period of time and context between client requests is maintained. These are also called WebSpeed transactions or a state-persistent transactions.

8.1.3 Advantages and Drawbacks of State-persistence

State persistent applications are advantageous because they:

- Allow a single database transaction to span multiple page requests and to be completely rolled back (undone) in case of error
- Minimize the amount of data that needs to be passed from one request to another
- Minimize database access (re-opening tables) for each request

However, there may also be considerable overhead associated with state persistent applications because they:

- Lock a WebSpeed Agent until a transaction terminates
- May tie up one or more database records until the transaction terminates

NOTE: All Web objects (state-aware or stateless) that access databases are subject to the rules that govern Progress record locking and database transactions. For more information on database transactions, see Chapter 9, "Controlling Database Transactions." For more information on record locking, see the *Progress Programming Handbook*.

8.2 Understanding WebSpeed Transaction Control

WebSpeed (or state-persistent) transactions allow you to maintain active context on a single WebSpeed Agent between requests by locking that Agent to a single Web client. To support WebSpeed transactions, WebSpeed uses an Agent control program. This program executes whatever Web object is specified in the WebSpeed URL for a request. In so doing, it verifies the status of the Web object in any active WebSpeed transaction, locks the Agent on behalf of the Web object that starts a WebSpeed transaction, and unlocks the Agent on behalf of any Web object that terminates the WebSpeed transaction.

To lock an Agent temporarily to a particular browser (thus starting a WebSpeed transaction), you run the function setWebState, before HTTP header output. This procedure performs two basic functions:

- It sets cookies that allow the WebSpeed Broker to identify the Web browser, the WebSpeed Agent, and the Web object that are bound in the transaction.

  The cookie settings also ensure that any subsequent WebSpeed request from the transaction-bound browser are serviced by the same locked WebSpeed Agent.

- It sets a specified time-out period (in minutes) for the invoking state-aware Web object. This period is the maximum time that the Web user has between subsequent transaction requests before the transaction automatically terminates. The locked Agent keeps track of this period for all state-aware Web objects that run in the current transaction.

  The transaction-bound Web client can make other Web requests, including WebSpeed requests using other Agents. However, the transaction clock runs while the browser is engaged with other requests. It can time-out if the user fails to continue the transaction within the specified time.

While the Agent is servicing requests for a WebSpeed transaction, requests made to the same URL by other Web clients cannot be serviced by that Agent until the current transaction terminates. The locked Agent continues to service both state-aware and stateless requests from the transaction-bound client, as long as those requests use the same WebSpeed Messenger.

8.2.1 How To Make a Web Object State-aware

Basically, you run the setWebState function to make a Web object state-aware, or to cancel the state-aware condition:

SYNTAX

```
setWebState (time-out).
```

time-out

The extent of the time-out interval stated in minutes:

- when the value is greater than zero, the Web object remains persistent for the specified number of minutes
- when the value equals zero, the Web object cookie is deleted and the Web object becomes stateless

NOTE: The setWebState function is a wrapper for the set-web-state method procedure. For more information about the syntax of setWebState or set-web-state, see the AppBuilder online help.

The setWebstate procedure must be run before the HTTP header output and should be run before any cookie functions are called. The following is an example of its use:

```
PROCEDURE output-header :

/*-----------------------------------------------------------------------

Purpose: Output the MIME header, and any "cookie" information

-----------------------------------------------------------------------*/

/* Check the status of the transaction. If the transaction is
complete, turn off state-aware. If the transaction is ongoing
set a 15 minute timeout period. */

IF state eq "complete":U

THEN setWebstate (0).
ELSE setWebstate (15).

/* This file is going to create a standard HTML page. */

RUN outputContentType IN web-utilities-hdl ("text/html":U).

END PROCEDURE.
```

For more detailed information about controlling WebSpeed transactions, see the section, "Implementing Transaction Control." For more information about time-outs, see the section, "Handling State-aware Time-outs."

8.2.2 Web Objects in Stateless and State-persistent Contexts

Figure 8–1 illustrates how a stateless Web object may run in a state-persistent context during a WebSpeed transaction and also run stateless within a different context.

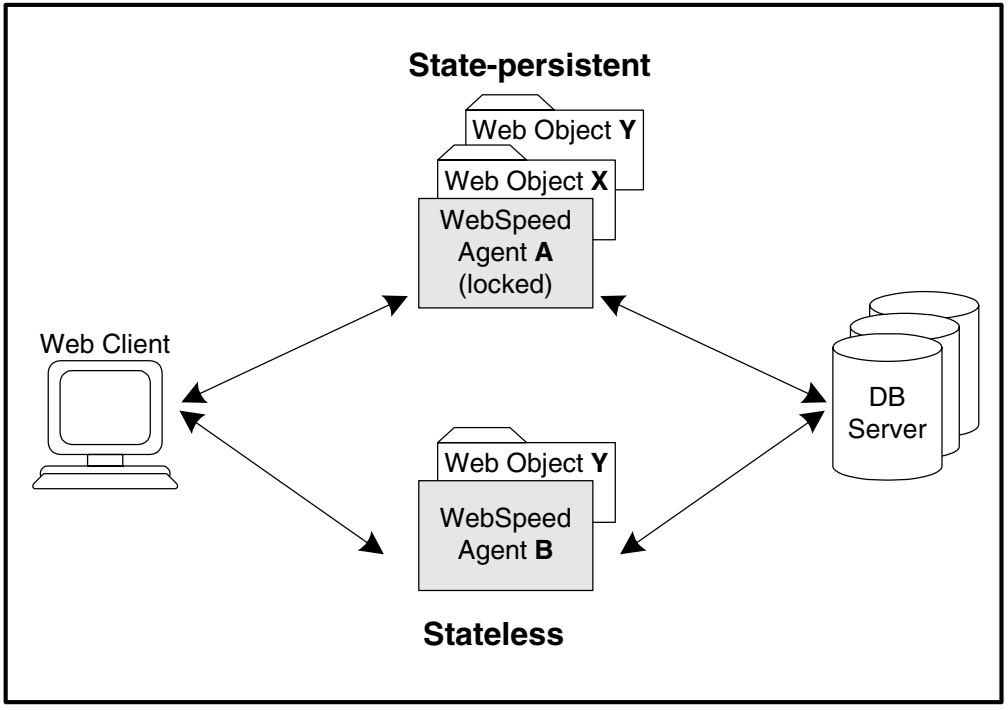


Figure 8–1: A Web Object Running Stateless and State-persistent

In this scenario, Web Object X is state-aware while Web Object Y is stateless. At the top of Figure 8–1, Web Object Y is called within the context of a WebSpeed transaction. Web Object Y is still stateless even though it executes on a locked Agent because it does not set a time-out period for itself in the transaction. However, Web Object Y does have potential access to the data context established by Web Object X, especially if Web Object Y calls a custom method procedure within Web Object X that returns data from this transaction context. In general, all Web objects that execute on a locked Agent participate in the same WebSpeed transaction, whether they are stateless or state aware.

If the Web Client makes a request to Web Object Y (possibly through a different Messenger and WebSpeed Broker), Web Object Y executes as a stateless Web object on WebSpeed Agent B. Here the Web object also runs stateless, but with no underlying transaction context available.

8.3 Primary and Secondary Web Objects

As shown in Figure 8–1, while you can construct a WebSpeed transaction that includes a combination of stateless and state-aware Web objects, practical management of these multiple requests can get tricky. The problem is that each successive Web object usually needs to know something about what occurred in the previous request, and it might not even know which Web object executed in the last request.

You can pass around additional cookies or use the URL to convey this information, but that wastes the advantage of the continuous context established by the WebSpeed transaction. You might, instead, use a single primary Web object that initiates the WebSpeed transaction and use secondary Web objects to provide additional pages in the transaction. This allows the secondary Web objects to share data with the primary Web object or to otherwise communicate through local data in the primary Web object.

A *secondary Web object* is a Web object that you execute from another Web object that is already servicing a request on the Agent. Typically, there is only one *primary Web object* that handles all requests and initiates the calls to all other secondary Web objects in the transaction.

You cannot use embedded SpeedScript files as primary Web objects because they cannot explicitly make themselves state aware. However, you can use embedded SpeedScript files as secondary Web objects in a state-persistent application. Thus, you must build any primary Web object using a CGI Wrapper Web object or an HTML-mapping Web object.

8.3.1 Running Primary and Secondary Web Objects

The fundamental requirement for a secondary Web object is that it generate or map a Web page so that the Web client must make the next transaction request back to the primary Web object.

The primary Web object typically sets the time-out period for the WebSpeed transaction. As the primary and secondary Web objects respond to requests from the client, the time-out counts down. The time-out does not reset unless the primary Web object explicitly resets it by calling setWebstate. Note that all secondary Web objects are implicitly state aware by virtue of the primary Web object's being state aware.

The primary Web object typically generates the HTTP header for the Web page. However, for embedded SpeedScript Web objects, the primary Web object must call the setWebstate method procedure to propagate the state-aware cookie for the next request service before calling any embedded SpeedScript Web object.

Figure 8–2 shows the relationship among a Web client, a primary Web object (Web Object A), secondary Web objects (Web Objects X, Y, and Z), and the transaction request cycle (solid arrows) in a state-persistent WebSpeed transaction. In this scenario one might imagine that Web Object A, in addition to controlling the secondary Web objects, defines shared variables that can be referenced by Web Objects X, Y, and Z.

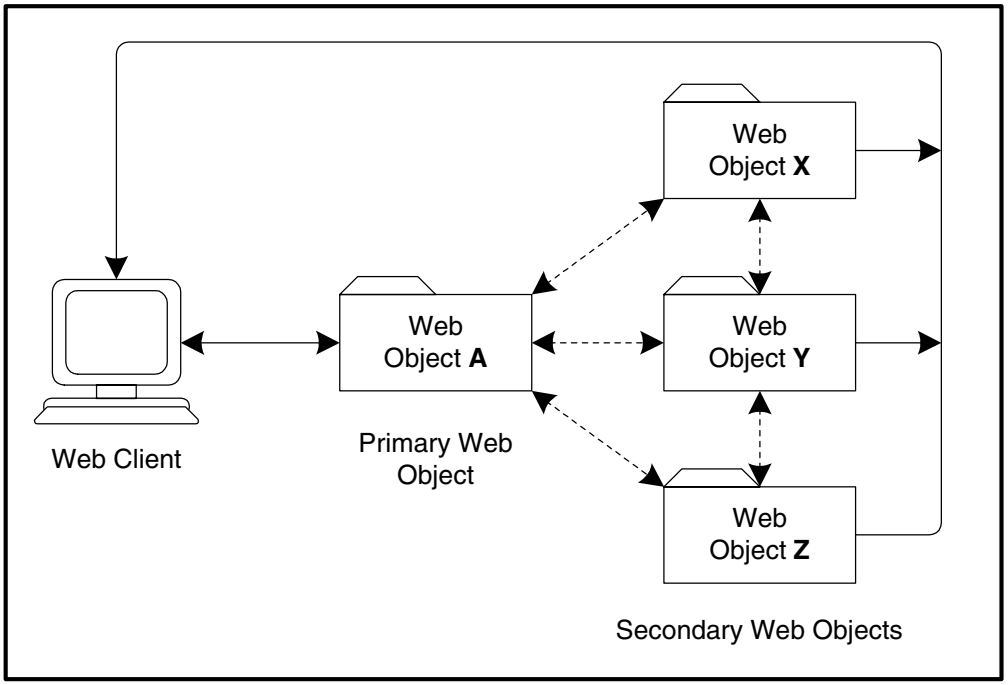


Figure 8–2: Primary and Secondary Web Objects in a WebSpeed Transaction

The request cycle shown in Figure 8–2 illustrates these points:

1. The Web client (browser) only calls the primary Web object (Web Object A) directly. The client can make repeated calls to the primary Web object, until the transaction times out.

2. Primary Web Object A can either:

   - Call a secondary Web object (Web objects X, Y, or Z).
   - Return a Web page to the client. Any links or references in the returned Web page point back to Primary Web Object A.

3. A secondary Web object can either:

   - Call some other secondary Web object.
   - Call the primary Web object.
   - Return a Web page to the client. Any links or references in the returned Web page point back to Primary Web Object A.

At some point Web Object A determines that the user is done, unlocks the Agent and possibly sends a Web page back to the user telling them that the transaction is terminated.

8.3.2 State-aware Web Objects and Persistence

You might have noticed that the scenario in Figure 8–2 seems to have Web Object A running again and again. This is not necessarily what happens.

Typically, the first time a state-aware Web object is executed, it initializes. The final step of initializing is to call its process-web-request method procedure and return to the caller after completing execution. However, the Web object typically remains in a suspended but active state known as *persistence*. In this persistent state, any of the Web object's method procedures, including process-web-request, can be executed by any other Web object.

The next time that the persistent Web object services a request, only its process-web-request procedure is executed (without the initialization code), and generally this procedure does not iterate or recur.

The primary Web object generally runs the secondary as a stateless Web object, which does not persist after execution. The usual course is to send a Web page for display that does not POST data back to the secondary Web object. However, the page might link back to the primary Web object for service by some other persistent Web object in the transaction.

8.4 Implementing Transaction Control

Web objects can be customized in order to control WebSpeed transactions. The methods for customizing Web objects include:

- Modifying the outputHeader procedure
- Modifying the process-web-request procedure
- Overriding procedures and functions
- Modifying the attribute list
- Modifying the user field list

In addition to these components, CGI Wrapper and HTML-mapping Web objects contain additional code sections that you can modify using the AppBuilder editor. For more information, see the "Transaction Control with CGI Wrapper Web Objects" section and the "Transaction Control with HTML-mapping Web Objects" section.

8.4.1 Modifying the outputHeader Procedure

The outputHeader procedure is the standard place to make a Web object state aware. It performs similar transaction management functions for both HTML-mapping and CGI Wrapper Web objects.

The standard WebSpeed template for this procedure contains a set of comments and a call to the output-content-type API function:

```
output-content-type ("text/html":U).
```

The call to output-content-type is required and it outputs the standard MIME header for Web pages.

The comments prior to output-content-type describe how to call setWebstate (or set-web-state) and set-cookie in order to make the Web object state aware.

The setWebstate function sets the WebSpeed transaction state for the Web object. In this example, the calling Web object is made state aware with a time-out period of five minutes:

```
setWebState (5).
set-cookie ("custNum":U, "23":U, today + 1, ?, ?, ?, ?).
output-content-type ("text/html":U).
```

For more information about the syntax of setWebstate, see the AppBuilder online help, or see the "How To Make a Web Object State-aware" section.

To make the Web object state aware, setWebstate creates the cookies that identify the WebSpeed Agent and the Web object. After a time-out, setWebstate kills the cookies.

The setWebstate function resides in *install-path*/src/web2/admweb.p. The Web object runs this super procedure persistently to provide access to many WebSpeed method procedures and API functions.

If you want to terminate the entire state-persistent WebSpeed transaction and unlock the Agent, you must execute setWebstate with the time-out period set to zero.

NOTE: WebSpeed supports one other standard place to call setWebState, when handling state-aware time-outs. For more information, see the "Resetting a Web Object's Time-out Period" section.

Although the setWebState function automatically creates the cookies that identify the WebSpeed Agent, and the Web object, you may want to create additional cookies to pass other information. The set-cookie API function, called after setWebState, allows you to generate additional application cookies. You can call it as many times as needed to define all your cookies.

For example, the following sets a cookie, custNum=23:

```
setWebState (5).
set-cookie ("custNum":U, "23":U, today + 1, ?, ?, ?, ?).
output-content-type ("text/html":U).
```

The cookie expires on the next day at midnight. Note that midnight is the default value if time is the unknown value (?).

For more information, see the "Passing Information Between Web Requests" section in Chapter 5, "SpeedScript." For syntax information on the set-cookie function, see the AppBuilder online help.

8.4.2 Modifying the process-web-request Procedure

The process-web-request procedure is the primary method for handling Web request input and Web page output for a Web object. For a state-aware Web object, this is the method procedure that the Agent control program calls to handle a request destined for that Web object.

The CGI Wrapper and HTML-mapping templates are commented for guidance in managing a generic Web request.

The CGI Wrapper template for this method procedure contains the required call to the outputHeader procedure and statements that output a skeleton Web page. Otherwise, the content of this procedure is entirely application-dependent.

The HTML-mapping template for this method procedure contains the required call to the outputHeader procedure and provides a complete default CGI GET and POST request framework in which to handle a Web request. This framework provides a default mechanism to move data between the mapped page and the Web object. You can (and generally must) modify both versions of process-web-request for your application. For more information, see the "Anatomy of process-web-request in HTML-mapping" section.

8.4.3 Overriding Procedures and Functions

WebSpeed provides mechanisms for customizing procedures and functions in a way that never touches the default source code. Thus, you can effectively override any procedure or function as required.

You can override a procedure or function by having the WebSpeed Section Editor insert a *local* version of the procedure or function in your Web object. This local copy can contain a call to the default version, or it can completely replace the behavior of the default version. (For an example local override of a procedure, see the "Handling State-aware Time-outs" section in this chapter.)

You can also create an include file containing the definitions for your procedures and functions. (Examples of include files reside in *install-path*`/src/web/method`.) You can then reference these include files almost anywhere you write SpeedScript code, whether in embedded SpeedScript files or SpeedScript procedures. For information on coding include files references in SpeedScript procedures, see the *Progress Language Reference*.

Whenever you invoke a procedure or function, WebSpeed first looks for a local version of the procedure or function to execute. If that is not found, WebSpeed executes the default version. The default version is found in a *super procedure* that runs persistently on the WebSpeed Agent.

A super procedure is an external procedure that runs persistently. Super procedures use the object oriented model for implementing and extending common behavior in applications. With super procedures, you can:

- Define standard behavior
- Execute standard behavior
- Override standard behavior with a local version
- Supplement standard behavior either before or after execution
- Overload or inherit multiple behavior classes

For more information about creating and using super procedures, see the *Progress Programming Handbook*. Also see the AppBuilder online help.

8.4.4 Modifying the Attribute List

An *attribute list* defines a set of name/value pairs that apply to a specific Web object executing on the Agent.

WebSpeed makes intensive use of the attribute list for each Web object to record critical information, such as the transaction state of the Web object. WebSpeed provides the set-attribute-list and get-attribute method procedures to access these lists.

Most WebSpeed access of these lists is hidden by other method procedures and API functions, such as setWebState. However, you can also set and retrieve your own Web object attributes using the set-attribute-list and get-attribute procedures. For more information, see the definition for these method procedures in *install-path*`/src/web/method/admweb.i`. Also see the AppBuilder online help for syntax information.

CAUTION: Do not modify the WebSpeed Web-State or Timeout-Period attributes directly using set-attribute-list, unless otherwise directed. The Agent control program relies on these settings to provide orderly support for Web object execution and control. For an example of a supported use of set-attribute-list, see the "Handling State-aware Time-outs" section.

8.4.5 Modifying the User Field List

A *user field list* defines a set of name/value pairs that apply to your entire application. The user field list is global to all Web objects executing on the Agent.

The function of the user fields is entirely application dependent. You can use it to maintain your own application states and to pass information between Web objects running on the same Agent for the same Web request (or WebSpeed transaction, if the Agent is locked).

To explicitly create and access user fields, WebSpeed provides the set-user-field and get-user-field API functions. For more information on these functions, see "Passing Information Between Web Requests" in Chapter 5, "SpeedScript." For an example of a user field used to manage a WebSpeed transaction, see the "Handling State-aware Time-outs" section in this chapter. Also see the AppBuilder online help for syntax information.

NOTE: The get-value API function looks first in the list of user fields. Generally, you use get-value instead of get-user-field to return user field values.

8.5 Transaction Control with Embedded SpeedScript Web Objects

Embedded SpeedScript Web objects have the simplest structure to support WebSpeed execution, as they are basically HTML files converted to Web objects. As such, they cannot execute the setWebState function to make themselves state aware. You can generally use embedded SpeedScript Web objects only as stateless or secondary Web objects in WebSpeed transactions. They always run non persistently both in and out of a WebSpeed transaction.

You can define an outputHeader method procedure in embedded SpeedScript files to output application cookies for the generated Web page. However, this embedded SpeedScript method procedure does not, by itself, support calling setWebState to make the Web object state aware.

For more information on creating outputHeader procedures in embedded SpeedScript files and coding for embedded SpeedScript Web objects, see Chapter 5, "SpeedScript."

8.6 Transaction Control with CGI Wrapper Web Objects

WebSpeed allows you to code a CGI Wrapper Web object by customizing well-defined code sections managed through the AppBuilder Code Section Editor. As such, CGI Wrapper Web objects support all of the techniques for starting and managing WebSpeed transactions.

In addition to the outputHeader and process-web-request procedures (described in the "Implementing Transaction Control" section), the default CGI Wrapper code sections include:

- **Definitions** — Where you can code preprocessor, parameter, and variable definitions that apply to the entire Web object.
- **Main Code Block** — The main-line of a Web object that performs initialization and executes process-web-request during initial execution and creation of the Web object. Generally, this is not an area that you modify unless you want to execute code once the first time a state-aware Web object is called. Even then, you generally write initialization code in a local initialize event procedure.

8.7 Transaction Control with HTML-mapping Web Objects

WebSpeed allows you to code an HTML-Mapping Web object by customizing well-defined sections of the code managed through the AppBuilder Code Section Editor.

In addition to the outputHeader and process-web-request procedures (described in the "Implementing Transaction Control" section), the default HTML-mapping code sections include:

- **htmOffsets**—(Read only) A procedure that automatically associates each form field with the corresponding field object (widget) in the Web object. The AppBuilder provides this section for reference only.
- **Definitions**—Where you can code preprocessor, parameter, and variable definitions that apply to the entire Web object.
- **Main Code Block**—The main-line of a Web object that performs initialization and executes process-web-request. Generally, this is not a section that you modify unless you want to execute code once the first time a state-aware Web object is called.
- **Control handlers**—Where you can override the default `web.input` and `web.output` control handlers for a specified field object. This code section appears only in HTML-mapping Web objects.
- **ADM methods (event procedures)**—Procedures that provide standard behavior for the HTML-mapping process-web-request and other sections of an HTML-mapping Web object. You often create overrides to these standard event procedures, for example displayFields, assignFields, or initialize.

8.7.1 Anatomy of process-web-request in HTML-mapping

The heart of an HTML-mapping Web object is process-web-request. Figure 8–3: shows how the default HTML-mapping version of process-web-request participates in a Web request.

This default model can apply whether the Web object is accessed by its URL from the browser or is called directly from another Web object. (The STEP numbers are taken from the default comments in the procedure.)

Each of the shaded boxes represents a method procedure, database action, or one of several event procedures that provides a basic request service:

- **outputHeader**—The method procedure that outputs the HTTP header.
- **inputFields**—An event procedure that moves all the HTML form element values received in a request to the corresponding field objects in the Web object. These field objects are initially defined by TagExtract when you create the Web object in the AppBuilder.
- **FIND/CREATE Database Record**—This is the standard point at which you might find or create a database record. This record provides the data for field objects mapped to database fields.
- **assignFields**—An event procedure that moves the values for update in the field objects to the corresponding variables and database fields in the Web object.
- **displayFields**—An event procedure that moves the values in variables and database fields to the corresponding field objects in the Web object.
- **enableFields**—An event procedure that makes some field objects sensitive to user input on the Web page. The value of any disabled field object mapped to HTML <INPUT> tags of type TEXT, HIDDEN, or PASSWORD becomes straight text in the output HTML, and does not return as input in the next request. Form element types for all other disabled field objects appear enabled, but their values do not get assigned by assignFields.
- **outputFields**— n event procedure that outputs a Web page to the Web server, merging all the field object values with the Web page according to the tags that are mapped to the Web object. This includes the handling of custom tags as well as form element tags. (The Web object reads the offset file at this point to merge data into the original HTML file that was used to generate the HTML-mapping Web object.)

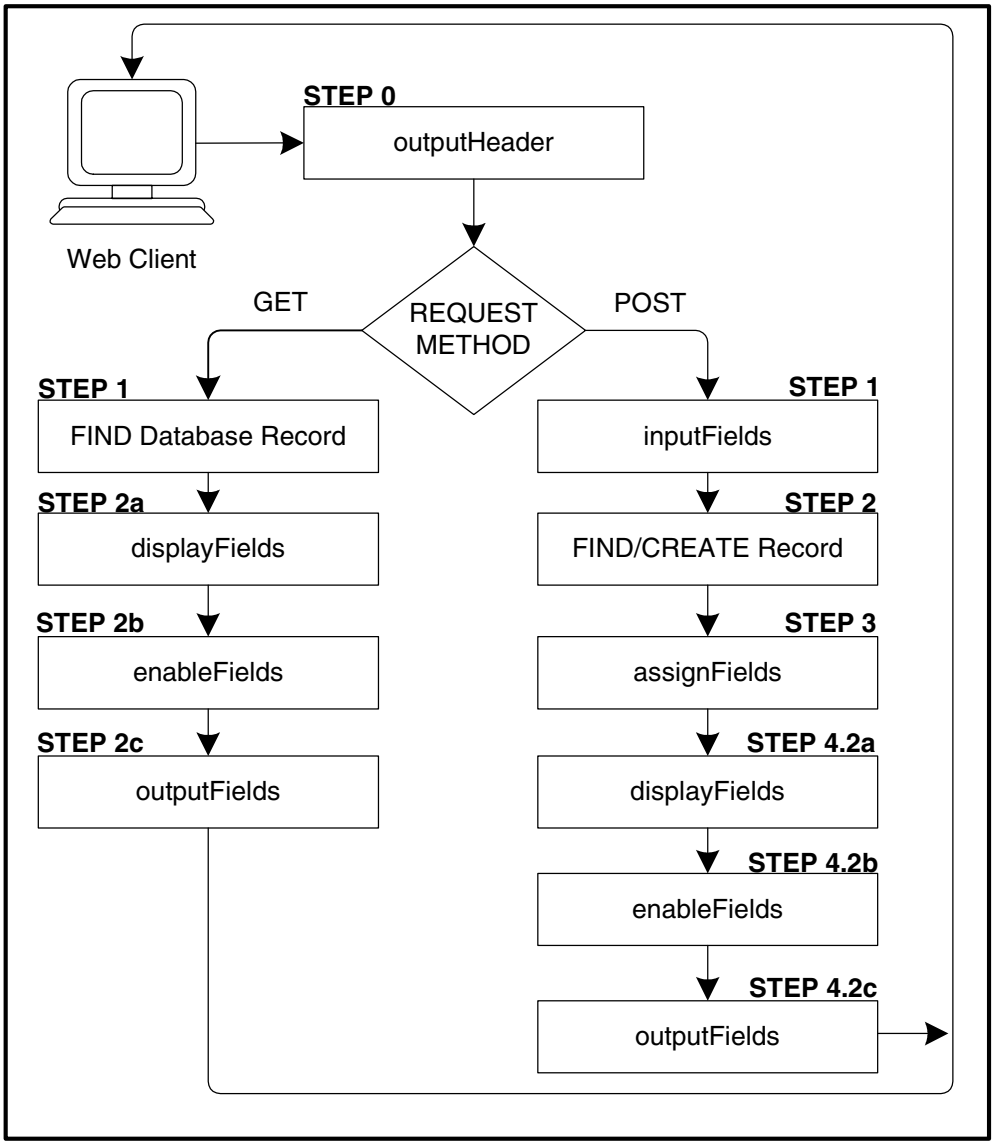


Figure 8–3: Standard HTML-mapping process-web-request Model

NOTE: Step 4.1 is missing from Figure 8–3. In the actual code (*install-path*/src/web2/template/html-map.w), this step describes how to simulate a Web request by calling another Web object. This is not a common action.

8.7.2 Modifying the Request Logic

The default process-web-request logic orders the procedure calls around a single test of the CGI REQUEST_METHOD variable. Aside from customizing the procedures themselves, you can (and often must) change the placement of these procedure calls in the logic, depending on your application. You can also add other tests for submit button values or other data that is returned with each request.

Typically, you have a GET when the browser user clicks a link or enters a URL to the Web object. Although you can create an HTML form that returns with another GET, limitations on the amount of data that a browser can return with a GET make a POST request the preferred choice. A GET passes data as part of the URL, while a POST passes data through the Agent's standard input.

However, as the comments in process-web-request indicate (see Chapter 4, "Sample Web Objects"), a GET request has other possible uses. If you want to return a different Web page from the one that was just posted, the Web object that handles the post has to call another Web object to provide the new page. You might do this after handling the current POST (STEP 3) and before you ordinarily begin to return the current form (STEP 4.2a). This is the missing STEP 4.1.

But when you call the new Web object, you want it called as if it were handling a CGI GET request, so it will return the new page as if for the first time. To simulate a GET request, WebSpeed allows you to assign the Web object REQUEST_METHOD variable to change the method (in this case from a POST to a GET) before you call the new Web object.

NOTE: While you can modify REQUEST_METHOD, this is not always a safe practice.

What about the data passed in with the actual POST? WebSpeed internally treats a GET or POST the same way. That is, when it retrieves data values input with the request, it always looks for data from a posted form in both the URL (retrieved from the CGI QUERY_STRING variable) and the standard input. Thus, you can change the request method for a called Web object to a GET and still allow the object to retrieve any data that came in with the previous POST.

NOTE: If you change the request method from a POST to a GET for a called Web object, it is up to you to change the request method back to a POST when the Web object returns to the caller.

You can also retrieve data independently of the apparent request method. For this purpose, WebSpeed provides the get-value( ) API function. This function returns the value for any named item, whether it comes from QUERY_STRING or the standard input.

Because Web objects do not care where their input comes from, you can write a WebSpeed application that bases its request logic solely on the values returned by the get-value( ) API function and ignore the inputFields method entirely or provide a local inputFields override.

NOTE: Unless you want to reset REQUEST_METHOD, or if the secondary Web object is state aware, you can run the secondary Web object directly.

8.7.3 Moving Data Through the HTML Mapping Web Object

To help understand the impact of the default logic in process-web-request and some of the effects you can expect from overriding it, Figure 8–4 shows where the default event procedures move data in a Web object.

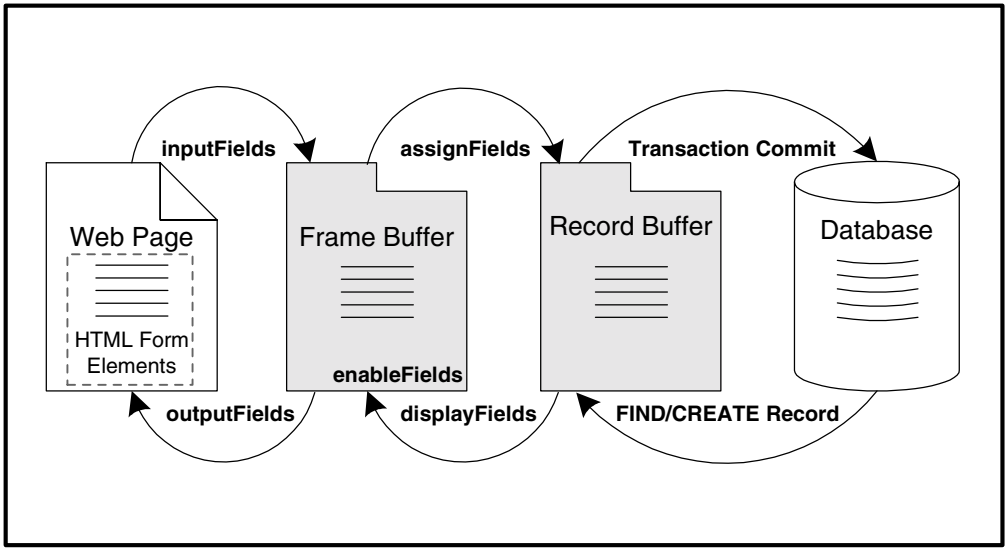


Figure 8–4: Data Flow Through an HTML-mapping Web Object

Frame Buffer and Record Buffer

The frame buffer is a Web object memory area for storing data that comes directly from HTML form input or that is being prepared for output to a Web page. This data is stored in a character string form that is compatible with its appearance in the Web page. Each data item is stored in a field object that corresponds to an HTML form element of the equivalent type. Thus, these field objects are windows into the frame buffer for each data item accessed by the Web object.

NOTE: The frame buffer in SpeedScript is equivalent to the screen buffer in the Progress 4GL. The difference is that for 4GL, each field object in the screen buffer supports interaction with a local keyboard and monitor. For SpeedScript, each field object in the frame buffer is a staging area for moving data to and from the Web. Any documentation provided with WebSpeed that references display values, screen values, or the screen buffer actually refers to the frame buffer and its data.

The record buffer is a memory area for storing Web object variables and database field values that are directly output to or input from a database. Each data item is stored in the record buffer according to the native SpeedScript data type (character, integer, and so on) that is defined for the corresponding variable or database field. The record buffer is the most efficient Web object storage and serves as the working storage for all SpeedScript computations and database I/O.

inputFields Data Movement

For a request that comes in with form input, inputFields retrieves the values for all input elements listed in `tagmap.dat` and places them in the corresponding field objects of the form buffer. inputFields does this by calling the default web.input control handler defined for each form element type.

NOTE: You can override the default web.input control handler in the New Section dialog box of the AppBuilder Section Editor.

assignFields Data Movement

Each field object in the form buffer corresponds by name to an HTML form element. Each field object also corresponds to a named variable or database field in the record buffer. (If you do not select a database field, the AppBuilder can define a variable with a name that is similar to the corresponding form element name, or you can define your own variable or buffer and set the field source in the AppBuilder to User.) assignFields moves the current values stored in the form buffer to the corresponding variables and fields in the record buffer, converting them to the appropriate data type, if necessary.

NOTE: assignFields only moves data from field objects for which you have set the Enable property to Yes (the default) in the field object Property Sheet in the AppBuilder.

Committed Transactions

At any point that the Transaction Agent or Web object commits the current database transaction (or subtransaction), any database records containing fields that were modified during the transaction (or subtransaction) are written to the database. This includes records modified by assignFields or by using SpeedScript directly. For more information on database transactions and subtransactions, see Chapter 9, "Controlling Database Transactions."

FIND Records

For a Web request, the typical database query returns one record (or one record per table in a join) from the database that provides the values for mapped field objects. Generally, you accomplish this using a single FIND statement. However, you might require other file I/O or calculations to provide the values for local variable or user field objects that are also mapped for the request. You can also retrieve records anywhere that is appropriate for your application, such as in a local override to an event procedure.

displayFields Data Movement

displayFields moves variable and field values from the record buffer to the form buffer, converting to formatted character strings, as appropriate. It also takes FORMAT options from the database or you can override these FORMAT options in the AppBuilder property sheet.

displayFields only moves data to field objects for which you have set the Display property to Yes (the default) in the field object Property Sheet in the AppBuilder.

enableFields Effects

enableFields modifies the SENSITIVE attribute of the appropriate field objects in the form buffer. If set to FALSE, text-oriented field objects are effectively prevented from receiving any input by the action of the default web.output control handler. For more information on these effects, see the code provided in *install-path*\src\web\support\webinput.p.

enableFields only sensitizes field objects for input for which you have set the Enable property to Yes (the default) in the field object Property Sheet in the AppBuilder.

outputFields Data Movement

outputFields performs the main tasks required to provide a Web page to the Web server. This includes merging data from the form buffer into the mapped Web page. outputFields does this by calling the web.output control handler defined for each mapped form element and custom tag that it encounters in the output Web page.

If the default field object is not enabled for input and the corresponding form element is an <INPUT> tag of type TEXT, HIDDEN, or PASSWORD, the method outputs the **text** of the value in place of the form element tag. Because the form element tag is effectively deleted and replaced by plain text, the value represented by that text can never be changed by the user or returned as a value in a subsequent request with the form. Thus, the next request from the Web page arrives as if the form element never existed, and the corresponding web.input control handler never sees any input for it. For all other form element types, such as RADIO, the default web.input returns whatever data is there to the form buffer, but it is never assigned to a program variable or database field.

The Display property setting for a field object has no effect on outputFields. Whatever data happens to be in the form buffer is output.

The output for a form element is entirely up to you, especially if you define your own field object mapping or override the default web.output control handler. For example, you might interpret the value for the form element, replacing or combining it on output with an HTML reference to an image (using the <IMG> tag).

You can conditionally enable and disable field objects for input during each request, but only if you first define the field object as enabled for input using the Properties Sheet in AppBuilder. That done, you can then set the SENSITIVE attribute of the field object, as required at any point between the default execution of enableFields and outputFields.

8.7.4 Creating Definitions

The following shows the Definitions section of the example Web object `w-custinf.w`. The boldface text shows what is added to the default provided by WebSpeed:

w-cstinf.w

```
/*------------------------------------------------------------------------

  File: w-cstinf.w

  Description: Find and update basic customer information in Sports2000.

  Input Parameters:
      <none>

  Output Parameters:
      <none>

  Author:

  Created:

------------------------------------------------------------------------*/
/*           This .W file was created with AppBuilder.           */
/*------------------------------------------------------------------------*/

/* ***************************  Definitions  ************************** */

/* Preprocessor Definitions ---                                         */

/* Parameters Definitions ---                                           */

/* Local Variable Definitions ---                                       */

DEFINE VARIABLE vButton AS CHARACTER NO-UNDO. /* Submit button value */
```

The initial comments block provides optional documentation on the entire Web object, including a description of any input and output parameters defined for it.

NOTE: You can specify input and output parameters only for Web objects that you call directly from other Web objects, using the RUN statement. For more information, see the "Running Procedures and Web Objects" section in Chapter 5, "SpeedScript."

You can add any parameter, variable, or preprocessor definitions that you need for your Web object in this section.

8.7.5 Modifying the Main Code Block

The following is a copy of the Main Code Block that executes when you run an HTML-mapping Web object:

```
/* *************************  Main Code Block  ********************** */

/*
 * Standard Main Code Block. This dispatches two events:
 *   'initialize'
 *   'process-web-request'
 * The bulk of web processing is in the procedure 'process-web-request'
 * elsewhere in this WebObject.
 */
{src/web2/hmapmain.i}
```

The include file `src/web2/hmapmain.i` contains the default code that is executed each time a stateless Web object is run and the first time a state-aware Web object is run:

```
/* The CLOSE event can be used from inside or outside the procedure to  */
/* terminate it.                                                        */
ON CLOSE OF THIS-PROCEDURE
   RUN destroy.

/* Now enable the interface and wait for the exit condition.            */
/* (NOTE: handle ERROR and END-KEY so cleanup code will always fire.    */
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON STOP    UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:

  /* Load the HTM handles etc. */
  RUN initialize.

  /* Process the current web event. */
  RUN process-web-request.
END.

/* Run the local/adm-destroy procedures, if the procedure is ending.    */
IF NOT THIS-PROCEDURE:PERSISTENT THEN RUN destroy.
```

This code first registers a call to the destroy event procedure as a SpeedScript trigger (ON CLOSE of the Web object).

NOTE: Initialization code is best done in a local initialize rather than changing the Main Code Block directly.

The default destroy procedure removes all traces of the Web object from memory. You can also override the destroy procedure to perform any other cleanup activities that you require before the Web object goes away. This is most useful in state-persistent applications to manage early termination of multi-page database transactions. For more information on database transactions, see Chapter 9, "Controlling Database Transactions."

The initialize event procedure performs a number of data assignments that are necessary before the Web object can execute. You can override this event procedure to add any one-time tasks that you require before process-web-request executes.

If you choose to add code directly to the Main Block before or after any initialization occurs and process-web-request executes, remember that this code executes only once for state-aware Web objects. Each additional time that the Transaction Agent or another Web object executes a state-aware Web object using the run-web-object method procedure, only the process-web-request procedure is executed for the state-aware Web object until it times-out. When it times-out, the destroy event procedure is executed.

8.7.6 Customizing Field Object Control Handlers

Field object control handlers implement the conversion between HTML and the field object for each form element and custom tag that is mapped to your Web object. This mapping is defined by the combination of your HTML Web page and the `tagmap.dat` file that you use for your application.

The following shows the default `tagmap.dat` file:

tagmap.dat

```
#Default data mappable fields
#
#  Do not move the first line below from its position.  The first line is
#  the default field type for fields missing TYPE=.

input,,text,fill-in,web/support/webinput.p
input,,checkbox,toggle-box,web/support/webtog.p
input,,hidden,fill-in,web/support/webinput.p
input,,password,fill-in,web/support/webinput.p
input,,radio,radio-set,web/support/webradio.p
select,/select,,selection-list,web/support/weblist.p
textarea,/textarea,,editor,web/support/webedit.p


#Custom Tag that can be used to support HTML Tables and 3rd Party controls
!--WSTAG,,,fill-in,web/support/tagrun.p

#Custom Tag that can be used to notify an application to output messages
#messages that have queued up using the queue-message function
!--WSMSG,,,fill-in,web/support/webmsg.p
```

The `tagmap.dat` file contains the HTML field definitions that specify the types of HTML form elements and custom tags used by all the Web objects in an application. WebSpeed provides a default `tagmap.dat` file that supports the common HTML form elements. You can add to or modify these field definitions as required. The default definitions include the HTML form elements that you can specify with the <INPUT>,<SELECT>, and <TEXTAREA> tags.

Each HTML field definition takes up a line in `tagmap.dat` that conforms to the following syntax:

SYNTAX

```
tag-name , [ closing-tag ] , [ typ-attribute ] ,
     field-object-type , [ utility-pathname ]
```

tag-name

The name that identifies the HTML tag. In an HTML form this is `input`, `select`, or `textarea`. For a custom tag, this is generally a name in the form of an HTML comment, such as `!--MyTag`. Using comments to define custom tags minimizes the chance of conflict with future versions of HTML. Note also that custom tags, unlike form elements, can appear anywhere in the HTML file.

closing-tag

The closing tag name for HTML tags that require them, such as `/textarea`.

type-attribute

The TYPE attribute for <INPUT> tags and any custom tags or future HTML tags that require a TYPE attribute such as `text` or `hidden`.

field-object-type

The type of SpeedScript field object to which this HTML or custom tag type is mapped. The supported SpeedScript field objects are EDITOR, FILL-IN, RADIO-SET, TOGGLE-BOX, and SELECTION-LIST. By default, these correspond to the HTML form elements to which they most closely resemble in both form and function. SpeedScript supports each field object with a unique set of capabilities provided by SpeedScript attributes and methods. For more information on these capabilities, see the entry for each field object (referred to online as a *widget*) in the *Progress Language Reference*.

utility-pathname

The pathname of a tagmap utility procedure file that contains the default web.input and web.output control handler procedures for this HTML or custom tag type. For the default field definitions, this path is relative to the PROPATH settings. When you customize a field object control handler, you are replacing the control handler functionality provided by the corresponding tagmap utility. If you do not specify a tagmap utility procedure, you must override web.input and web.output in the AppBuilder to do equivalent work.

NOTE: Any line in `tagmap.dat` that begins with a pound sign (#) is a comment.

The Web page for the example ncust-wo.w. illustrates how the field definitions correspond to the form elements and custom tagsin an HTML file:

ncust-wo.htm

```
<HTML>

<BODY>

<FORM ACTION="ncust-wo.w" METHOD="post">

<CENTER>

<P><B>Enter portion of a<BR>customer last name:</B><BR>

<INPUT TYPE=text NAME="cust-prompt" SIZE=16 >

<INPUT TYPE=submit NAME="CustSearch" VALUE="Search">    </P>

</CENTER>

<P>Please enter some starting portion of a customer name

even if only a single letter.</P>

<SELECT Name="matching-cust-names" Size=10> </SELECT>

<INPUT TYPE=submit NAME="CustDetail" VALUE="Show Detail" >

<P>
Name:    <INPUT TYPE=text NAME="Name" SIZE=25 > <BR>
Phone: <INPUT TYPE=text NAME="Phone" SIZE=25 > <BR>
Comments: <TEXTAREA NAME="Comments" ROWS=6 COLS=60 ></TEXTAREA> <BR>
<BR>
Country:<BR>
USA <INPUT TYPE=radio NAME=Country VALUE=1><BR>
Other <INPUT TYPE=radio NAME=Country VALUE=2> <BR><BR>
Has Orders: <INPUT TYPE=checkbox NAME="HasOrders">
</P>

<INPUT TYPE=submit NAME="CustUpdate" VALUE="Update" >

</FORM>

</BODY>

</HTML>
```

When the TagExtract utility generates the offset file for this Web page using the default `tagmap.dat` file, it generates this offset file:

ncust-wo.off

```
/* HTML offsets */
htm-file= /working-directory-path/ncust-wo.htm
version= AB_v19r1

field[1]= "cust-prompt,INPUT,text,fill-in,11,1,11,45"
field[2]= "matching-cust-names,SELECT,,selection-list,21,1,21,53"
field[3]= "Name,INPUT,text,fill-in,26,10,26,47"
field[4]= "Phone,INPUT,text,fill-in,27,8,27,46"
field[5]= "Comments,TEXTAREA,,editor,28,11,28,63"
field[6]= "Country,INPUT,radio,radio-set,31,5,31,43"
field[7]= "Country,INPUT,radio,radio-set,32,7,32,45"
field[8]= "HasOrders,INPUT,checkbox,toggle-box,33,13,33,50"
```

NOTE: You can interrupt offset generation at any point in an HTML file by inserting <!--TagExtractSuspend--> at that point. You can then resume offset generation by inserting <!--TagExtractResume--> at a following point in the file.

The offset file records each mappable tag (field) in order of its occurrence in the HTML file. For each tag (`field[6]`), it records the NAME attribute value (`Country`), tag TYPE (`INPUT`), HTML visualization (`radio`), the corresponding SpeedScript field object type (`radio-set`) from `tagmap.dat`, and four integers that record the starting line, starting character, ending line, and ending character of the tag specification in the HTML file (`31,5,31,43`). The only relevant information missing from this file is the location of the tagmap utility procedures. WebSpeed continues to get this information both during development and at run time from the `tagmap.dat` file (`web/support/webradio.p`).

NOTE: TagExtract understands UNIX pathnames in `tagmap.dat` (`web/support/webradio.p`) as well as MS-DOS pathnames (`web\support\webradio.p`). However, when you move your application to UNIX, you can only use UNIX pathnames. Therefore, `tagmap.dat` uses UNIX pathnames by default.

For many applications, the default tagmap utilities work well with the default tags. However, you can replace the default `web.input` or `web.output` control handler for any tag using the control handler templates shown in the following:

web.input

```
/*------------------------------------------------------------------------
  Purpose:     Assigns form field data value to frame screen value.
  Parameters:  p-field-value
  Notes:
------------------------------------------------------------------------*/
  DEFINE INPUT PARAMETER p-field-value AS CHARACTER NO-UNDO.

  DO WITH FRAME {&FRAME-NAME}:

  END.

END PROCEDURE.
```

web.output

```
/*------------------------------------------------------------------------
  Purpose:     Output the value of the field to the WEB stream
               in place of the HTML field definition.
  Parameters:  p-field-defn
  Notes:
------------------------------------------------------------------------*/
  DEFINE INPUT PARAMETER p-field-defn AS CHARACTER NO-UNDO.

  DO WITH FRAME {&FRAME-NAME}:

  END.

END PROCEDURE.
```

Each template passes the same parameter for every tag. For web.input, this is the current input value associated with the tag, expressed as a character string. This value is actually the output of the get-field( ) API function for the HTML field name.

For web.output the parameter is the full HTML tag specification for the field from the source HTML file. If you want the output value, you must access the SpeedScript SCREEN-VALUE attribute of the field object associated with the tag. You can reference the attribute using the full name of the field object. The field object name always has the same name as the NAME attribute for the associated tag (replacing embedded spaces with underscores (_) and removing any illegal SpeedScript characters). Thus, for the Name field in `ncust-wo.htm` you can access the form buffer value using Name:SCREEN-VALUE.

If the SpeedScript field object is a radio set, web.output executes once for each radio item. A second parameter (the item number) is also passed into the control handler.

Placing your code inside the default DO block ensures that the object is properly referenced in its SpeedScript frame. Note also that, if the field object maps to a database field, the object name can be prefixed by the database and table names separated by periods (for example, sports2000.Customer.Name), depending on your AppBuilder settings.

Finally, when creating custom tags, you can create your own tagmap utility procedures, such as `webinput.p`. However, it is very common to rely on the web.input and web.output control handlers in the Web object itself, instead of defining a separate utility procedure for the tag. This works when the custom tag maps to a standard SpeedScript field object like fill-in.

8.8 Handling State-aware Time-outs

A Web object time-out occurs when the time out period specified in a Web object's Web-Timeout attribute has expired. When this happens, an Agent that was LOCKED changes its status from LOCKED to AVAILABLE and destroys the Web object in the process (as long as there are no other state-aware objects still active).

However, if a Web object times-out, but the Agent is still LOCKED, the object's "Web-State" moves to "Timed-Out", but the object is not destroyed. You can still access the Web object while it is timed-out until the Agent changes status to AVAILABLE. If you do not want this behavior, you can add the following code to process-web-request:

```
RUN get-attribute ("Web-State":U).
IF RETURN-VALUE eq "Timed-Out"
THEN DO:
    RUN HtmlError IN web-utilities-hdl ("Object has timed out!").
    RETURN.
END.
```

In general, when the user submits a request from a page returned by a now timed-out Web object, the request contains HTML that includes a stale WSEU cookie. The Transaction Server returns the following message:

```
The Web object to which you were attached has timed out. Please start again.
```

If you receive a timed-out message, you can take one of three actions:

1. Set the Web-Timeout-Handler attribute of the state-aware Web object, which allows you to run another Web object (.w) or WebSpeed procedure (.p) when the state-aware Web object times-out.

2. Reset the time-out period for a state-aware Web object by overriding the default adm-timing-out event procedure.

3. Rely on the default time-out message from the WebSpeed Transaction Server.

8.8.1 Using a Web Object Time-out Handler

To use a time-out handler for a Web object, set the Web-Timeout-Handler attribute to the name of a procedure or Web object that you want to run when the state-aware Web object times-out. You only need to set this attribute once, so the best place to set the attribute is in the Main Code Block or a local initialize event procedure for the Web object.

Set the attribute using the set-attribute-list method procedure:

```
RUN set-attribute-list IN THIS-PROCEDURE ('Web-Timeout-Handler =
mytimeout.w':U).
```

The procedure specified as the Web-Timeout-Handler is stored both as the handle of the state-aware Web object and as part of its Web object cookie. This supports the situation where the state-aware Web object times-out and is deleted. Even if the Web object is deleted, the Web-Timeout-Handler can still be retrieved from the cookie.

Here is an example of HTTP headers generated by WebSpeed for a Web object containing the Web-Timeout-Handler attribute:

```
Set-Cookie: WSEU=demeter:5604:22744:0; path=/cgi-bin/timeoff.sh
Set-Cookie: task.w=97,login.w ; path=/cgi-bin/timeoff.sh
Set-Cookie: employee.ssn=000000000; path=/cgi-bin/timeoff.sh
Content-Type: text/html
```

As you can see, the Web-Timeout-Handler (login.w) is appended to the end of the Web object cookie, which contains the value of the SpeedScript UNIQUE-ID procedure attribute for the state-aware Web object.

After a time-out occurs, WebSpeed retrieves the time-out handler name from the cookie of the timed-out state-aware Web object during the next Web request and runs the procedure by that name. If the time-out handler procedure is itself an HTML-mapping Web object, note that WebSpeed executes the section of process-web-request that handles the appropriate request method, usually a POST.

8.8.2 Resetting a Web Object's Time-out Period

Normally, when a Web object times-out, the timingOut event procedure runs and changes the object's state to "Timed-Out". To reset the time-out period after a state-aware Web object has timed-out, create a timing-out override to this event procedure using the Procedure Editor in the AppBuilder. In this local override procedure, call the setWebState function to make the Web object state-aware with a new time-out period.

To ensure that the Web object times-out at some point, allow the local override procedure to call the default timingOut procedure when your application no longer allows the time-out to be reset:

```
PROCEDURE timingOut :
/*------------------------------------------------------------------------
  Purpose:     Override standard ADM method
  Notes:
------------------------------------------------------------------------*/

  /* Code placed here will execute PRIOR to standard behavior. */

  /* Dispatch standard ADM method.                                 */
  IF get-user-field("Application-State") = "No-More-Changes" THEN
    RUN SUPER.
  ELSE
   setWebState (60).

  /* Code placed here will execute AFTER standard behavior.    */

END PROCEDURE.
```

In this example, the application must set the "Application-State" user field to "No-More-Changes" when it is ready to allow the time-out to proceed. Otherwise, when any time-out for this Web object occurs, it remains state-aware for another 60 minutes.

CAUTION: It is important to verify the code you use to conditionally execute the default timingOut procedure. If it never executes, the Agent that is running the state-aware Web object remains locked because the time-out period is always reset. You can always force the object to time-out by running setWebState in the outputHeader procedure with a 0 time-out setting.

8.8.3 Allowing WebSpeed to Handle Time-outs

Your application does not have to handle a state-aware Web object time-out. In this case, WebSpeed outputs the default message (see the "Handling State-aware Time-outs" section in this chapter).

9

Controlling Database Transactions

This chapter describes how WebSpeed supports database transactions, including:

- A definition of database transactions
- Understanding the scope of database transactions
- SpeedScript components and database transactions
- Database transactions in applications
- Determining when database transactions are active
- Transaction system mechanics
- Efficient database transaction processing
- Multi-page database transactions

9.1 What Is a Database Transaction?

A *database transaction* (DB transaction) is a unit of work that is either completed as a unit or undone as a unit. Proper database transaction processing is critical to maintaining the integrity of your databases.

Suppose you are entering new customer records into your database and are entering the 99th customer record. If your machine goes down, are the first 98 records you entered lost? No, because WebSpeed:

- Keeps the first 98 records in the database
- Discards the partial 99th record

This is just one simple scenario. Suppose the procedure was updating multiple tables. You want to make sure that WebSpeed saves any completed changes and discards partial changes in all tables.

System failures are just one kind of error. There are other kinds of errors that can occur while a procedure is running. Regardless of the kind of error you are dealing with, data integrity is all important. *Data integrity* means that WebSpeed only stores completed data in the database. WebSpeed uses database transactions to automatically handle this processing.

For any WebSpeed application that updates a database, you must consider at what point you want a database transaction to begin and how many page requests you need it to last. In other words, how much of a database update do you want to roll back at one time in the event of an error, exception, or incorrect data input.

9.1.1 DB Transactions in Code

The terms *physical transaction* and *commit unit* refer to the same concept as the WebSpeed database transaction. For example, in the previous scenario where you are adding customer records, each customer record you add is a database transaction. In Figure 9–1, each order-line you create is a database transaction.

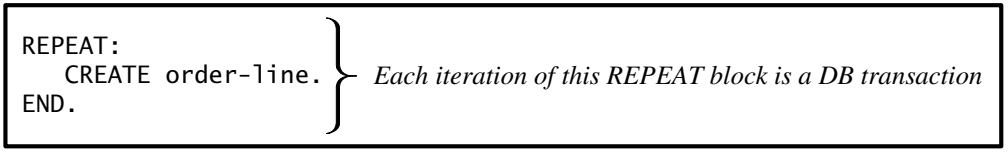


Figure 9–1: Database Transaction Definition

The database transaction is undone (or backed out) if:

- The system goes down (or crashes)
- The user presses the STOP button in a browser
- A state-aware Web object times-out

In either of these cases, WebSpeed undoes all work it performed since the start of the database transaction, as shown in Figure 9–2.

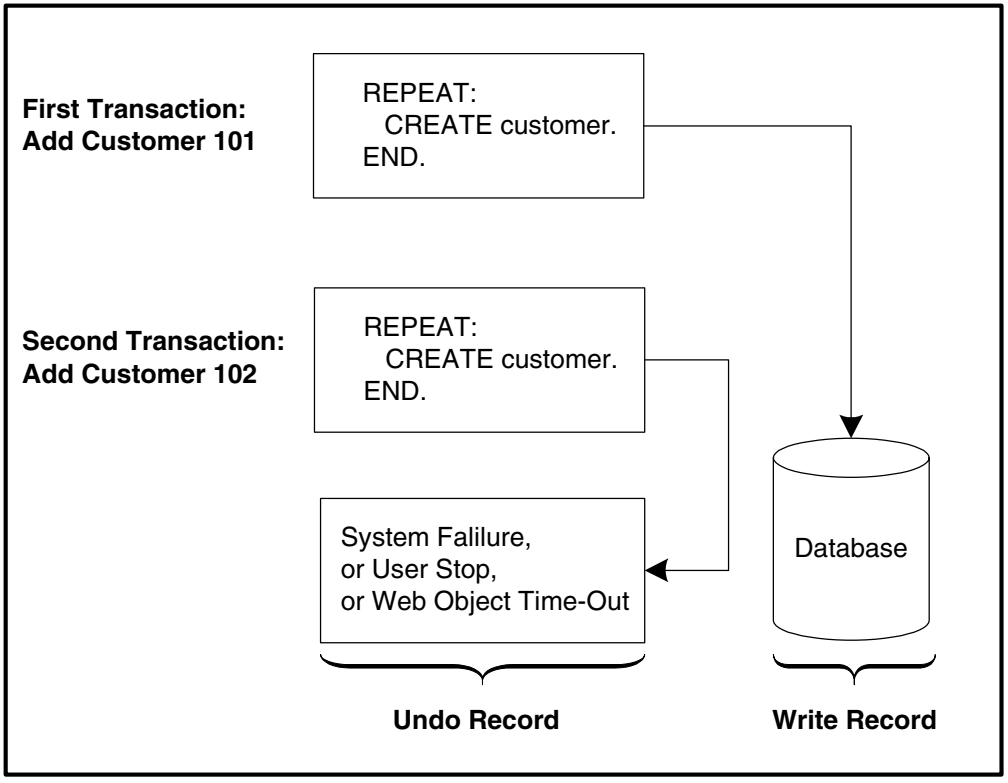


Figure 9–2: Database Transaction Undo Processing

So far, you have seen how a database transaction can be useful in a situation that involves only a single table. Database Transactions take on additional importance when you make database changes in multiple tables or records.

9.1.2 All-or-nothing Processing

Suppose a customer calls to change an order for four sweat bands to nine sweat bands. This means you must make two changes to your database:

- First, you must look at the customer's order and change the quantity field in the appropriate order-line record.
- Second, you must change the value of the allocated field in the record for that item in the item table.

What if you changed the quantity field in the order-line record and are in the midst of changing the allocated field in the item record, when the machine goes down? You want to restore the records to their original state. That is, you want to be sure that WebSpeed changes both records or changes neither.

Figure 9–3 shows this scenario for a database similar to Sports2000 (same tables, different data).

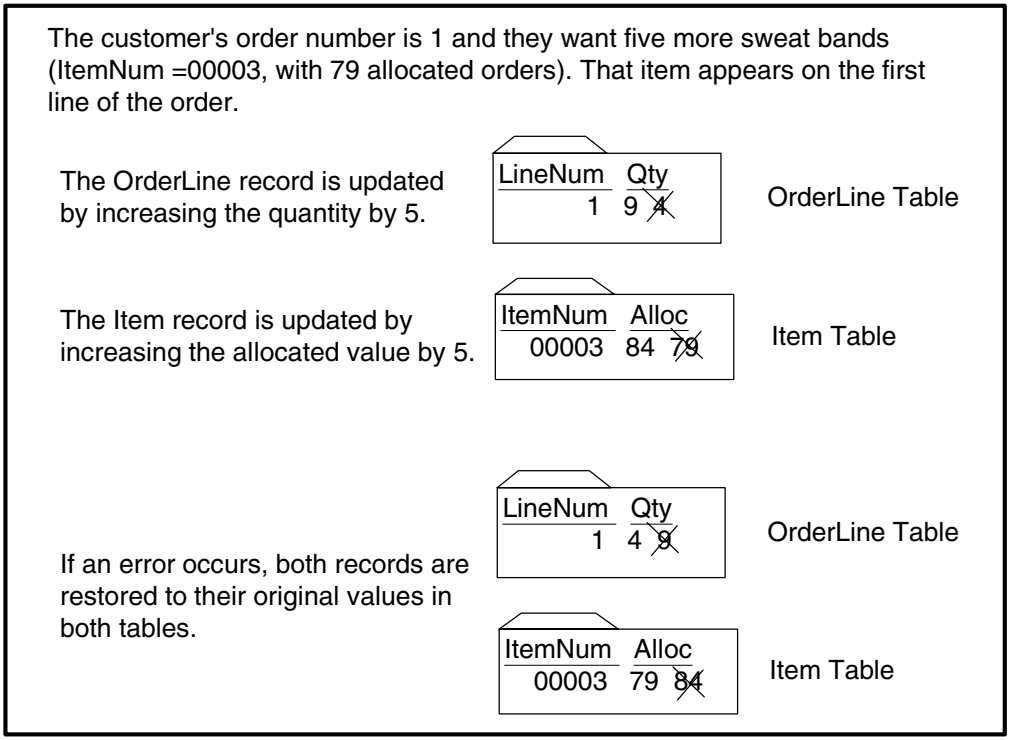


Figure 9–3: Database Transaction Involving Two Tables

9.2 Understanding the Scope of DB Transactions

How does WebSpeed know where to start the database transaction and how much work to undo or back out? The following *transaction blocks* start a database transaction if one is not already active:

- Any block that uses the TRANSACTION keyword on the block statement (DO, FOR EACH, or REPEAT).
- A procedure block, trigger block, and each iteration of a DO ON ERROR, FOR EACH, or REPEAT block that directly updates the database or directly reads records with EXCLUSIVE-LOCK. You use EXCLUSIVE-LOCK to read records in multi-user applications.

Directly updating the database means that the block contains at least one statement that can change the database. CREATE, DELETE, and UPDATE are examples of such statements.

If a block contains FIND or FOR EACH statements that specify EXCLUSIVE-LOCK, and at least one of the FIND or FOR EACH statements is not embedded within inner transaction blocks, then the block is directly reading records with EXCLUSIVE-LOCK.

Note that DO blocks do not automatically have the transaction property. Also, if the procedure or database transaction you are looking at is run by another procedure, you must check the calling procedure to determine whether it starts a database transaction before the RUN statement.

Once a database transaction is started, all database changes are part of that transaction, until it ends. Each user of the database can have just one active transaction at a time. The procedure in Figure 9–4 has two blocks: the procedure block and the REPEAT block.

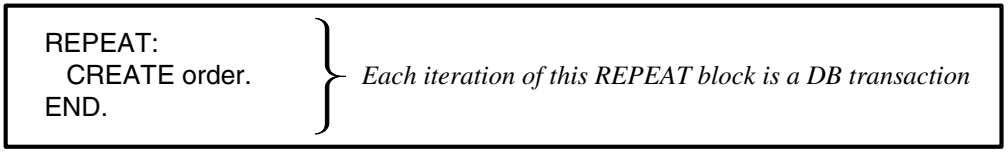


Figure 9–4: Database Transaction Scope

The procedure block has no statements directly in it that are not contained within the REPEAT block. The REPEAT block contains a CREATE statement that lets you add order records to the database. Because the REPEAT block is the outermost block that contains direct updates to the database, it is the transaction block.

At the start of an iteration of the REPEAT block, WebSpeed starts a database transaction. If any errors occur before the END statement, WebSpeed backs out any work done during that transaction.

Note that data-handling statements that cause WebSpeed to automatically start a database transaction for a regular table will not cause WebSpeed to automatically start a transaction for a work table or temporary table.

Figure 9–5 shows a procedure with multiple transactions.

NOTE: This example uses the GetField method procedure where WebSpeed usage recommends the get-field( ) API function. See the online AppBuilder Help for more information.

```
DEFINE VARIABLE line-info AS CHARACTER.
DEFINE VARIABLE i AS INTEGER.
REPEAT:
    CREATE order.
    FIND customer OF order.
    i = 1.
    REPEAT:
       CREATE order-line.
        RUN GetField in web-utilities-hdl
            (INPUT "line" + TRIM(STRING(i)),
             OUTPUT line-info).
        IF line-info = ""
            THEN LEAVE.
        ELSE
            DO:
            CREATE order-line.
            order-line.order-num = order.order-num.
            {&DISPLAY} order-line.order-num.
            ASSIGN line-num = i.
            i = i + 1.
            END.
    END.
END.

FOR EACH salesrep:
    {&DISPLAY} sales-rep rep-name.
    ASSIGN credit-limit = credit-limit * 1.10.
END.
```

DB Transaction

DB Transaction

Figure 9–5: Multiple DB Transactions in a Procedure

This procedure has four blocks:

- **Procedure block—**There are no statements in this block, so WebSpeed does not start a database transaction at the start of the procedure.
- **Outer REPEAT block**—The outermost block that directly updates the database (CREATE order WITH 2 COLUMNS). Therefore, it is a transaction block. On each iteration of this block, WebSpeed starts a database transaction. If an error occurs before the end of the block, all work done in that iteration is undone.
- **Inner REPEAT block—**Directly updates the database but it is not the outermost block to do so. Therefore, it is not a transaction block. It is, however, a subtransaction block. Subtransactions are discussed later in this chapter.
- **FOR EACH block—**An outermost block that directly updates the database (UPDATE region). Therefore, it is a transaction block. On each iteration of this block, WebSpeed starts a database transaction. If an error occurs before the end of the block, all work done in that iteration is undone.

9.2.1 Subtransactions

A subtransaction is started when a database transaction is already active and WebSpeed encounters a *subtransaction block*. If an error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. Subtransactions can be nested within other subtransactions.

The following are subtransaction blocks:

- A procedure block that is run from a transaction block in another procedure.
- Each iteration of a FOR EACH block nested within a transaction block.
- Each iteration of a REPEAT block nested within a transaction block.
- Each iteration of a DO TRANSACTION, DO ON ERROR, or DO ON ENDKEY block. (These blocks are discussed later in this chapter.)

Table 9–1 shows when database transactions and subtransactions are started.

Table 9–1: Starting Database Transactions and Subtransactions

Type of Block	Transaction Not Active	Transaction Active
DO transaction FOR EACH transaction REPEAT transaction Any DO ON ENDKEY, DO ON ERROR, FOR EACH, REPEAT, or procedure block that directly contains statements that modify database fields or records or that read records using an EXCLUSIVE-LOCK.	Starts a transaction	Starts a subtransaction
Any FOR EACH, REPEAT, or procedure block that does not directly contain statements that either modify the database or read records using an EXCLUSIVE-LOCK.	Does not start a subtransaction or a transaction	Starts a subtransaction

Note that data handling statements that cause WebSpeed to automatically start a transaction for a database table do not cause WebSpeed to automatically start a transaction for a work table or temporary table.

9.2.2 Controlling Where DB Transactions Begin and End

You might find that, because of the kind of work a procedure does, you want to start or end database transactions in locations other than those WebSpeed automatically chooses. You know that WebSpeed automatically starts a database transaction for each iteration of four kinds of blocks:

- FOR EACH blocks that directly update the database
- REPEAT blocks that directly update the database

- Procedure blocks that directly update the database
- DO ON ERROR or DO ON ENDKEY blocks that contain statements that update the database

A database transaction ends at the end of the transaction block or when the transaction is backed out for any reason.

Sometimes you want a database transaction to be larger or smaller depending on the amount of work you want undone in the event of an error. You can explicitly tell WebSpeed to start a database transaction by using the TRANSACTION option with a DO, FOR EACH, or REPEAT block header:

- DO TRANSACTION:
- FOR EACH TRANSACTION:
- REPEAT TRANSACTION:

When you explicitly tell WebSpeed to start a database transaction, it starts a transaction on each iteration of the block regardless of whether the block contains statements that directly update the database. Of course, WebSpeed does not start a database transaction if one is already active.

9.3 SpeedScript Components and DB Transactions

Database transactions have varying effects on different SpeedScript components. This section describes the effects on:

- Subprocedures
- File I/O
- Program Variables

9.3.1 Subprocedures

If you start a database transaction in a main procedure, that transaction remains active even while the main procedure runs called procedures. In Figure 9–6, `p-txn11.p` runs `p-txn11a.p` within a database transaction.

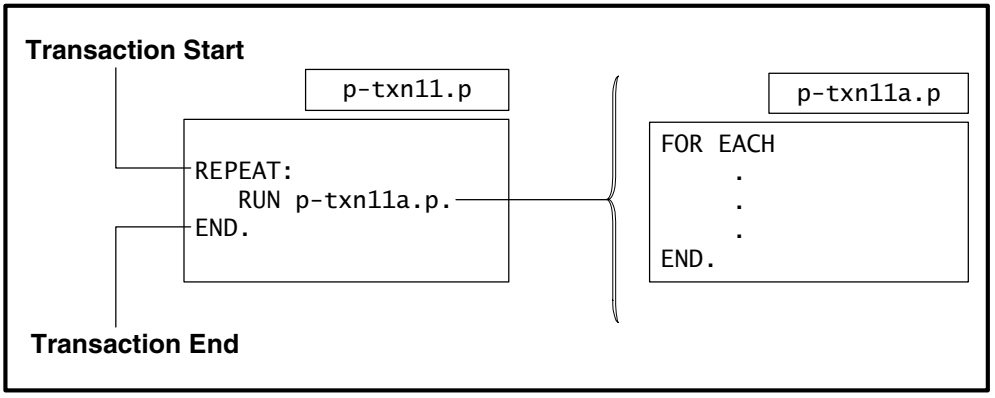


Figure 9–6: DB Transactions and Subprocedures

The REPEAT block `p-txn11.p` procedure is the transaction block for that procedure: it contains a direct update to the database. The database transaction begins at the start of each iteration of the REPEAT block and ends at the end of each iteration. That means when the `p-txn11.p` procedure calls the `p-txn11a.p` procedure, the transaction is still active. So all the work done in the `p-txn11a.p` subroutine is part of the transaction started by the main procedure, `p-txn11.p`.

If a system error occurs while you are processing orders for a customer, WebSpeed undoes all the order processing work you have done for that customer, as well as any changes you made to the customer record itself.

9.3.2 File Input

You must be especially careful when you process database transactions that read input from a text file, such as when you populate a database. If a crash occurs, the active database transaction is backed out of the database, but you will not automatically know how far processing proceeded in the text file. To handle this, you must do one of the following:

- Restore the database and rerun the procedures that were running when the system failed.
- Run all data input processes as a single database transaction (this has record locking implications as well as implications in terms of the size of the before-image file for the transaction).

- Have a way to determine how many of the input data lines were processed and committed to the database so that you do not rerun lines that were successfully processed by a procedure.

  One technique for doing this is to save the filename and the last line number in a database table, since changes in this status information can be synchronized with the corresponding database updates.

9.3.3 Program Variables

You have read quite a bit about how database transactions are backed out and how database changes are undone. But what happens to work done with variables?

Any changes made to variables in a database transaction or subtransaction block are undone whenever a database transaction or subtransaction is backed out. The variables are restored to the values they had at the beginning of the transaction or subtransaction that is undone. Variables specifically defined as NO-UNDO are not undone in this case. However, changes to variables made outside a database transaction are never undone since only transaction and subtransaction blocks can be undone.

Although backing out of variables is useful in many cases, there is a certain amount of overhead associated with undoing variables in database transactions. If you are doing extensive calculations and have no need for undo services, then consider using the NO-UNDO option on variables and arrays.

9.4 DB Transactions in Applications

In applications, database transactions are affected by distribution of components. For applications that include multiple databases, WebSpeed expands any transactions to include all the database involved. For applications that include Progress AppServers (distributed applications), each application component (Transaction Agent and AppServer) controls separate database transactions.

9.4.1 Multi-database Applications

In a multi-database application, you generally do not have to code any additional database transaction handling. Multi-database transactions are handled in much the same way that single-database transactions are handled. The WebSpeed two-phase commit mechanism ensures that any database transaction is either committed to all affected databases or to none. You should check to see that all necessary databases are connected before you start a database transaction.

Two-phase Commit

During a database transaction, WebSpeed writes data to one or more databases as program control passes through database update statements in the transaction block. At the end of a transaction block, WebSpeed tries to commit the changes to the databases. WebSpeed uses a two-phase commit protocol to commit the changes to the databases. In the two-phase commit protocol, WebSpeed polls all the databases affected by the transaction to see if they are reachable.

In the first phase of the two-phase commit, WebSpeed checks whether it can reach each database and makes the appropriate validation checks for each database. If any one of the databases is unreachable or the validation checks fail for a database, WebSpeed backs out the transaction and returns the databases to their pretransaction states using the before-image files. If all of the databases are reachable and their validation checks succeeded, WebSpeed commits the changes to the databases.

For more information on two-phase commit, see the *Progress Database Administration Guide and Reference*.

Checking Database Connections

If you want to test database connections prior to entering a database transaction, use the CONNECTED function:

sample7

```
IF CONNECTED("db1") AND CONNECTED("db2")
THEN RUN txnblk.p. /* transaction block */
ELSE RUN HTML-Error IN web-utilities-hdl
        ("Unable to perform transaction").
```

NOTE: The `txnblk.p` procedure exists only for illustration.

You should connect to all databases affected by a database transaction prior to entering a transaction block. As a general rule, do not execute a database connection in a transaction block. The database connection overhead could lock records in other databases affected by the transaction for a considerable length of time. A database connection failure also causes a database transaction error. WebSpeed defers DISCONNECT statements in a database transaction until the transaction completes or is undone.

For more information about connecting and disconnecting databases, see the CONNECT and DISCONNECT statements in the *Progress Language Reference*.

9.4.2 Distributed Applications

When a requesting application with an active transaction runs a remote procedure, Progress does not propagate the transaction to the remote procedure. Rather, the remote procedure acts as if it is the first procedure of the application, and follows the normal SpeedScript rules for starting and terminating transactions. If a requesting application and a remote procedure connect to the same database, each database connection comprises a separate transaction.

For more information on remote procedures, see *Building Distributed Applications Using the Progress AppServer*.

9.5 Determining When DB Transactions Are Active

You can use the TRANSACTION function to determine whether a database transaction is active in a procedure. The TRANSACTION function can help you identify the transaction and subtransaction blocks within a procedure.

You can get more information about database transaction activity by using the LISTING option on the COMPILE statement. See the *Progress Language Reference* for more information on the COMPILE statement and the TRANSACTION function.

9.6 Transaction System Mechanics

So far, this chapter has explained the actions WebSpeed takes for different kinds of errors and how you can override those actions and specify your own. But there is another side to what WebSpeed is doing during database transactions and subtransactions. The next two sections summarize the mechanics of database transactions and subtransactions. For more information on the mechanics of DB transactions and their administration, see the *Progress Database Administration Guide and Reference*.

9.6.1 Database Transaction Mechanics

During a database transaction, information about all database activity occurring during that transaction is written to a *before-image* (BI) file. WebSpeed maintains one BI file for each database. The information written to the before-image file is carefully coordinated with the timing of the data written to the actual database table. That way, if an error occurs during the database transaction, WebSpeed uses this before-image file to restore the database to the condition it was in before the transaction started. Information written to the before-image file is not buffered. It is written to disk immediately.

Space in the before-image file is allocated in units called clusters. WebSpeed automatically allocates new clusters as needed. (You can use the PROUTIL TRUNCATE BI utility to set the cluster size.) After all changes associated with a cluster have been committed and written to

disk, WebSpeed can reuse the cluster. Therefore the disk space used by the before-image file depends on several factors including the cluster size, the scope of your database transactions, and when physical writes are made to the database (.db) file.

9.6.2 Subtransaction Mechanics

If a database transaction is already active and WebSpeed encounters a DO ON ERROR, DO TRANSACTION, FOR EACH, REPEAT, or procedure block, WebSpeed starts a subtransaction. All database activity occurring during that subtransaction is written to a *local-before-image* (LBI) file. WebSpeed maintains one LBI file for each WebSpeed Transaction Agent. If an error occurs during the subtransaction, WebSpeed uses this local-before-image file to restore the database to the condition it was in before the subtransaction started. WebSpeed uses the local-before-image file to back out variables and to back out subtransactions in all cases when an entire database transaction is not being backed out.

Note that the **first** time a variable is altered within a subtransaction block, **all** of the variables in the procedure are written to the LBI file as a record.

Because the local-before-image information is not needed for crash recovery, it does not have to be written to disk in a carefully synchronized fashion as does the before-image information. This minimizes the overhead associated with subtransactions. The local-before-image file is written using normal, buffered I/O.

The amount of disk space required for each user's LBI file depends on the number of subtransactions started that are subject to being undone.

9.7 Efficient DB Transaction Processing

Here are a few guidelines to improve the efficiency of database transaction processing procedures:

- If you are doing extensive calculations with variables and you do not need to take advantage of undo processing for those variables, use the NO-UNDO option when defining the variables.
- If you are processing array elements, process them in a DO WHILE block rather than in a REPEAT WHILE block. That way, you will not start a separate database transaction or subtransaction for each array element.
- When the logic of your application permits, do as much processing as possible directly at the database transaction level rather than creating subtransactions. This principle should not restrict the way you implement your application, but you should use it whenever it is convenient.

9.8 Multi-page DB Transactions

In addition to database transactions controlled by a single Web object or procedure, WebSpeed supports the option of database transactions that last for multiple state-aware Web requests. That is, if you begin a WebSpeed transaction in your application, you have the option of starting a database transaction on the Transaction Agent that lasts for the duration of the WebSpeed transaction or until you terminate the transaction by explicitly undoing, retrying, or committing the transaction. Furthermore, as long as at least one state-aware Web object remains active, you can continue to start and terminate these multi-page (Agent) database transactions.

9.8.1 Managing Multi-page DB Transactions

The mechanism for controlling multi-page database transactions consists of two web-utilities-hdl method procedures that you can execute within process-web-request:

- **set-transaction-state (INPUT *t-state*)**, where *t-state* is a character string that can take these values:
  - "START[-PENDING]"—Begin a multi-page transaction when this request service ends (when the Transaction Agent returns to a LOCKED state or is no longer in a BUSY state).
  - "UNDO[-PENDING]"—Undo the current multi-page transaction when this request service ends.
  - "COMMIT[-PENDING]"—Commit the current multi-page transaction when this request service ends.
  - "RETRY[-PENDING]"—Undo the current multi-page transaction when this request service ends and immediately start a new database transaction.

  NOTE: You can drop the "-PENDING" suffix when you set a transaction state.
- **get-transaction-state**, which returns the current transaction state, `t-state`, as the value of the RETURN-VALUE SpeedScript function, including these values:
  - "NONE"—There is no multi-page database transaction active on this Agent.
  - "START-PENDING"—A multi-page transaction will be started on this Agent before the next request service (no Agent transaction is currently active).
  - "ACTIVE"—There is an active multi-page database transaction on this Agent.

- "UNDO-PENDING"—The current multi-page transaction will be undone by the start of the next request service.
- "COMMIT-PENDING"—The current multi-page transaction will be committed by the start of the next request service.
- "RETRY-PENDING"—The current multi-page transaction will be undone and a new multi-page transaction started before the start of the next request service.

As you can see, get-transaction-state can return two more states than you can set with set-transaction-state. This is because when you start or terminate a multi-page transaction, there is no effect until the next request service in the current Agent WebSpeed transaction. Thus, a transaction state of "NONE" means both that there is no multi-page database transaction current and none has been requested. Likewise, a transaction state of "ACTIVE" means both that there is an active multi-page transaction and no termination request has been issued.

Conversely, if the transaction state is "UNDO-PENDING" or "COMMIT-PENDING", there is an active transaction that will terminate by the next Web request service on this Agent. "RETRY-PENDING" means there is an active transaction that will be thrown away and there will be a new active transaction by the start of the next request.

Note that you cannot set any state at any time. Some transaction states preclude the setting of others and an error is returned when you violate theses requirements:

- "START" can only be set if the current transaction state is "NONE".
- "UNDO", "RETRY", and "COMMIT" can only be set if the current transaction state is "ACTIVE".

Thus, you cannot change a transaction state that you have previously set in the same request service.

Although you cannot start a multi-page transaction when no WebSpeed transaction is active on the Agent, you receive no error for trying. The Agent (`web-disp.p`) resolves the transaction request with the "NONE" state by default.

9.8.2 Working with the Multi-page Transaction Example

WebSpeed provides an HTML-generating Web object that allows you to start and terminate multi-page database transactions. This is `tran-tst.w` in the WebSpeed examples directory. This Web object allows the user to view and update the CustNum and name fields of the Customer table. There are buttons to change the object between State-Aware and State-Less. There are also buttons to set the Transaction-State to Start, Undo, Retry, and Commit.

Here is a set of sample steps:

1 ♦ Run `tran-tst.w` in your favorite broker.

2 ♦ Press the State-Aware button.

3 ♦ Press the Start button. This request sets up `web-disp.p` to start the transaction FOR THE NEXT REQUEST. Any updates made in this request are OUTSIDE THE TRANSACTION.

4 ♦ Press the Refresh button. This shows that the transaction state has changed to "ACTIVE".

5 ♦ Type **1** in the CustNum field and type **New Name for Skiing** in the Name field.

6 ♦ Press the Update button.

7 ♦ Type **2** in CustNum and **New Name for Frisbee** in Name.

8 ♦ Press the Update button.

9 ♦ Continue similarly Steps 7 and 8 as long as you want.

10 ♦ Press the Undo button. This tells `web-disp.p` to undo the changes. However, the changes will not actually be undone until this request ends. The customer list at the bottom (which is created in the current request) still shows the changed names.

11 ♦ Press the Refresh button. Now the name changes are undone.

Feel free to play with this code, accepting and undoing changes. Pay extra attention to when the transaction states become active.

Other things to try include:

1 ♦ Press Start when the Web object is stateless.

2 ♦ Press Start twice (that is, where the second press occurs when the transaction state is "ACTIVE").

3 ♦ Press Undo, Retry, or Commit when there is no transaction active.

9.8.3 Guidelines for Usage

Here are some helpful hints on using multi-page database transactions:

- Make sure there is at least one state-aware Web object.
- RUN set-transaction-state ("START"), and:
  - Note that any changes made in the same object while the state is "START" are not in the transaction.
  - The Web object that turns on the transaction should be "display-only," and not do any database updates.
- Continue processing requests in this Agent. Remember that **all** requests handled by the Agent (even stateless ones) are done within this transaction.

  If the user has a locked Agent and that Agent has a multi-page transaction, then every thing done by that user will be in the transaction. Be very careful about this. The user might go back to another page or try to run another Web object. Even if these are not state aware themselves, if the Agent runs them, then their changes will be within the transaction.
- When the user is done, set the transaction state to either "UNDO" or "COMMIT". Also set all the Web objects to stateless. Remember that setting all the objects to stateless automatically performs an "UNDO" unless you explicitly commit the transaction ahead of time.
- If you decide to set "UNDO" or "RETRY", remember that changes made within the same Web request will also be undone. That is, code such as:

```
RUN set-transaction-state IN web-utilities-hdl ('UNDO').
FIND LAST Customer.
Customer.Name = 'Sic gloria transit'.
{&OUT} 'Name is ' Customer.Name.
```

  This code shows the new name for the customer. However, the change will be undone with the rest of the transaction.
- Remember to consider the case of an Agent (WebSpeed transaction) time-out. This will also force an "UNDO" of the multi-page transaction.

Finally, Figure 9–7 shows a flow chart of the WebSpeed multi-page transaction model. It summarizes most of what is described in this section and can serve as a visual guide to the actual process of managing these database transactions.

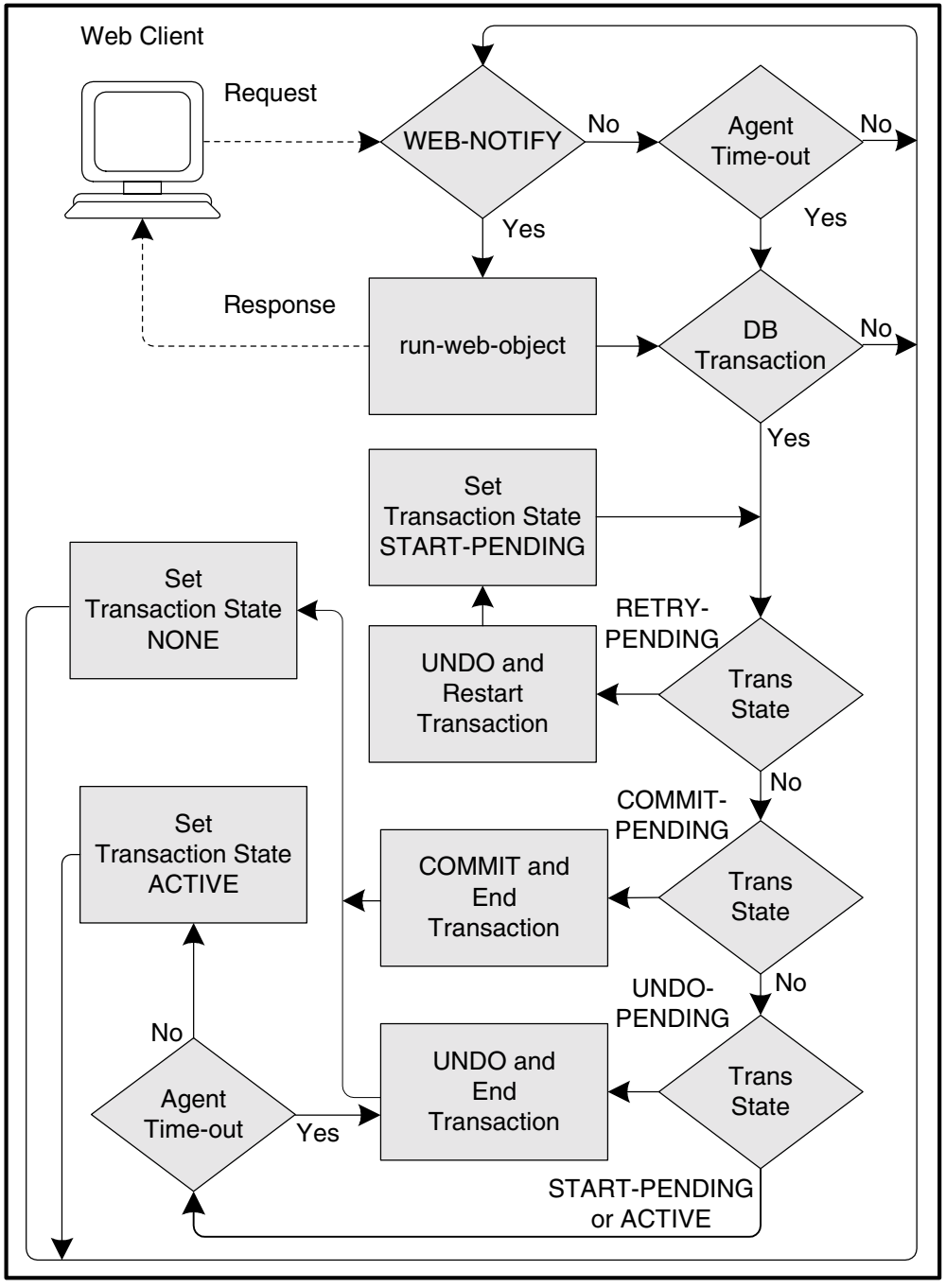


Figure 9–7: WebSpeed Multi-page Transaction Model

10

Running Applications

This chapter describes how to set up, run, and debug WebSpeed applications. This includes an overview of the configuration and setup procedures. For more information on these procedures, see your *WebSpeed Installation and Configuration Guide*.

The following sections include information on:

- Setting up an application environment
- Starting and stopping an application
- Debugging WebSpeed applications

10.1 Setting Up an Application Environment

This section describes how to set up WebSpeed on the machine that will run the Broker and Agents. Remember that your Web server machine and the machine that runs the Broker and Agents can be the same machine. However, you might want to use more than one machine to run WebSpeed, depending on the size of your WebSpeed application and the computing power of your Web server machine.

10.1.1 Creating a Working Application Directory

You must create a working directory for your WebSpeed application. Usually, this directory is where you place all of your WebSpeed application files-your .r, .w, .off, and .htm files. This directory also serves as the working directory for your Broker and Agents (this is the recommended configuration, but is not required). The workDir parameter, which you set in the ubroker.properties file or with Progress Explorer, tells the Broker which directory to use as its working directory. This is also the working directory for all of the Agents in the Broker's Agent pool. For more information about setting the working directory, see the *WebSpeed Installation and Configuration Guide*.

NOTE: You can also create other subdirectories in the working directory, or create procedure libraries and other directories in your PROPATH for when you are ready for deploying a production application.

The Broker's working directory is added automatically to the PROPATH of the Broker and Agents; the Broker and Agents use the PROPATH to locate your application files. The Broker's working directory is not explicitly named in the PROPATH, but is referenced using a dot (.). The dot is interpreted as the current working directory by each process that searches the PROPATH. If you choose to place your application files in a different directory, you must add that directory to the PROPATH or reference the file with a subdirectory in its pathname. This is an extra step that you can avoid by placing all of your files in the same working directory. For more information about setting the PROPATH, see the *WebSpeed Installation and Configuration Guide*.

10.1.2 Moving Application Files to Appropriate Directories

After you create a working directory, move the appropriate application files into the working directory. However, you must move some files to an appropriate Web server directory, such as the document root. Here is a list of common files that you must place in appropriate directories:

Web objects and procedures

Whenever you create a runnable procedure using the AppBuilder, you create a Web object. Sometimes, you create Web objects that are mapped to HTML files, but this is not required to dynamically generate HTML pages. Either way, during development, you must move all of your WebSpeed procedure files (.w, .p, or .i) into your application PROPATH. This ensures that the Agents can find and execute your WebSpeed procedures. If you precompile all of your procedures (creating .r files), you can remove the source files when you actually deploy the application. For more information, see the "Compiling Web Objects" section.

HTML and other static files

As described earlier in this document, some of the HTML files and images that you use in your Web presentation might never be processed by WebSpeed. These HTML files are served as static HTML pages that never change throughout the life of your Web presentation. You must place these files on your Web server machine so that your Web server can serve them directly to the Web user. However, if your WebSpeed application uses HTML files to dynamically generate HTML pages, you must make the HTML files visible to the Broker and Agents. The easiest way to achieve this is to place them in your application working directory (the Broker's default directory) or elsewhere in PROPATH.

Java class files

You should place Java class files in a subdirectory of the Web server root directory. Then you can reference the subdirectory in the CODEBASE attribute of the APPLET tag and the Java class file in the CODE attribute of the APPLET tag.

For example, if the Java class file, `myclass.class` is in a subdirectory of the Web server root directory called `java_classes`, the APPLET tag might look like the following:

```
<APPLET CODE="myclass.class" CODEBASE="/java_classes"></APPLET>
```

JavaScript files

You should place JavaScript (.js) files in a subdirectory of the Web server root directory. Then you can reference the relative path in the SRC attribute of the SCRIPT tag.

For example, if the JavaScript file, `myscript.js` is in a subdirectory of the Web server root directory called `javascript`, the SCRIPT tag might look like the following:

```
<SCRIPT LANGUAGE="JavaScript" SRC="/javascript/myscript.js"></SCRIPT>
```

tagmap.dat

If you modify the default `tagmap.dat`, place a copy of the modified `tagmap.dat` into your working directory. If you do not modify the file, you do not have to copy the file because the default `tagmap.dat` is used.

Offset files

An offset file (.off) is created whenever you use the AppBuilder to map an HTML file to a Web object. Agents use the offset file information to dynamically generate an HTML page. The purpose of the offset file is to provide the location of the HTML form fields in the HTML file. You ensure that the Agents can find your offset files by placing them in your working directory or in the directory with your running Web objects (or r-code). If an offset file is not current for its HTML-mapping Web object, the Agent generates a new offset file from your mapped HTML file and the available `tagmap.dat` file.

NOTE: Some of the above files are ASCII files and some are binary; therefore, when you transfer the files, make sure you use the ASCII or Binary transfer mode appropriate to each set of files.

10.1.3 Compiling Web Objects

Typically, you compile Web objects as you develop them in the AppBuilder. For example, when you save a source file in a Procedure Window, a dialog comes up that asks if you want to compile the file.

There might be occasions when you want to compile multiple files. You can compile single or multiple files in the WebTools File Tools interface. From the AppBuilder main menu, select Tools→ WebTools. Then select File Tools from the left frame of the browser. The File Tools interface has a list box from which you can select multiple files, and a compile button.

You can also use the Application Compiler, which is described in the *Progress AppBuilder Developer's Guide*. However, the Application Compiler is does not compile HTML files directly, so you could not use it on embedded SpeedScript sources.

10.1.4 Making Your Application Secure

A Web user can potentially run any procedure file that is accessible from an Agent's PROPATH. This includes any procedures in your application working directory and any procedure files that are relative to the *install-path* directory. You probably do not want to give the Web user this ability. For example, the procedure can compile and run WebSpeed programs typed in by a Web user. If you make this procedure available to a Web user, you essentially give that Web user complete control over an Agent. A number of SpeedScript statements would even give the Web user access to your operating system's file structure.

There are a number of things you can do to avoid this possibility. One approach is to start your Agents with the Run Run-time Client (-rr) startup parameter. This parameter ensures that Agents can only run precompiled code. This allows you to leave uncompiled procedures on the PROPATH without concern that they can be run from a Web browser. However, this approach does not allow you to take advantage of WebSpeed's compile-time flexibility. Depending on how you want to write your application, this might be important.

You can also use the check-agent-mode API function to allow some routine to work for Development but not for Production. For more information on the environment options, see the UNIX `webspeed.cnf` or see the *WebSpeed Installation and Configuration Guide*.

Yet another approach is to move any procedures off of the PROPATH that you do not want a Web user to run. For example, if you do not want a Web user to run the `runscrpt.w` procedure, then you must move it into a directory that is not included on the PROPATH and **is not relative** to the PROPATH.

NOTE: None of the WebSpeed tools can run in Production mode.

10.2 Starting and Stopping an Application

As described in the *WebSpeed Installation and Configuration Guide*, you must first make a data source available before starting a Transaction Broker and Agents for your application. This involves starting one or more Progress database servers or DataServers. When you shut down your application, be sure to shut down the Broker and Agents first, then shut down your databases. Otherwise, your Agents lose their database connections prematurely and might require operating system intervention to complete the shutdown.

10.3 Debugging WebSpeed Applications

This section describes some debugging techniques that you can use to debug WebSpeed applications. This allows you to see various types of information related to each Web request while running your application. The debugging information is appended to each Web page that your application generates. Application debugging must be enabled to use it. If debugging is enabled, a WebSpeed global variable, debugging-enabled, is set to TRUE. For more information on Environment or Application Mode and Debugging parameters, see your *WebSpeed Installation and Configuration Guide*.

10.3.1 Adding debug=on to URL

The WebSpeed Transaction Server enables you to view information about your Web requests. You can see this information by adding an argument (debug=on) to the URL that you normally use to access your WebSpeed application. For example, if your URL is normally:

```
http://webserver/cgi-bin/webapp.cgi/login.w
```

you can turn on all of the debugging options by adding the debug=on argument to the end of the URL. For example:

```
http://webserver/cgi-bin/webapp.cgi/login.w?debug=on
```

This runs your `login.w` program as usual. However, when WebSpeed finishes executing your program, it runs a debugging procedure. The output from this procedure appears in the browser directly below the output of your application. By default, WebSpeed runs the debugging procedure after your application program to minimize any problems with the debugging output affecting the application itself.

10.3.2 Persistent Debugging Uses Cookies

This debugging capability works best if you are using a browser that supports Persistent State Cookies. The browsers Netscape Version 3.2 and later and Microsoft Internet Explorer Version 3.0 and later support cookies. A number of other browsers also support this useful feature. It is very easy to determine if your browser supports cookies or not. This is outlined in the "Debugging and Administrative Options"section.

Without cookies, to use the debug=on feature, you would need to add debug=on to every URL in your application to ensure that this feature stayed on for subsequent page requests. By using a cookie, WebSpeed stores the current debug settings and uses them until you explicitly turn them off.

There are several sections in the output to the WebSpeed Request Information page. Each of these sections has an associated option to turn it on independently of the other options.

10.3.3 Calling the Virtual Debug Web Object

Each WebSpeed installation has access to a virtual Web object, debug, that you can invoke to start an application debugging session. You invoke debug by appending it like a Web object to the Messenger in your WebSpeed URL:

```
http://webserver/cgi-bin/webapp.cgi/debug
```

Invoking this URL from your browser causes the WebSpeed debugging Administration Form to become active during application execution. This form provides numerous options to control your debugging session.

10.3.4 Debugging and Administrative Options

This section, which appears at the bottom of the debug output, provides you with an interface to all the known debugging options, allowing you to turn them on or off. You can also implement your own application-specific debugging options by adding them to the Current debug options fill-in field or specifying them in the URL. The listings described in this section are supported by *install-path*/src/web/support/prinval.p.

The value displayed after Debugging Cookie WSDebug is the current value of the Cookie. Due to the way Cookies work, this value always lags behind the value displayed by the Current debug options fill-in, by one request.

To synchronize the Cookie with the debug options, change the debugging options as desired and click the appropriate button to set them. This sets the WSDebug Cookie appropriately. Then click the Reload submit button **on the form** (not your browser) to reload the page. The values for Current debug options and WSDebug Cookie are then identical. You can then scroll the page back to the top and run your application as usual. Whatever debugging options you set remain active until explicitly changed. However, these debugging settings are set specific to the URL of your application. The WSDebug Cookie is set with the path based on the value of the AppURL variable (see the "Miscellaneous Variables" section for more information about AppURL). This causes the browser to only send the Cookie when visiting any URL that starts with the same leading path.

If your browser does not support Persistent State Cookies, then the value of the WSDebug Cookie remains blank.

Some other options are:

- ON = turn on all of the above.
- OFF = turn off all debugging.
- ALL = synonym for the "on" option.

You can also specify multiple debugging options in the URL. Any of the options listed above can be combined. For example, to test the state-aware features, you might specify:

```
http://webserver/cgi-bin/webapp.cgi/webapp.w?debug=agent,cookies,http
```

This returns the output from a file named webapp.w followed by the specified debugging sections. If you want the Debugging and Administration form, the admin option must be specified explicitly unless the generic options "on" or "all" are used. You can also specify this list in the Current debug options fill-in field of the Debugging and Administrative Options form.

Turning off debugging is similar to turning it on. Click the "off" link on the Debugging and Administrative Options form or specify debug=off as an argument in the URL. Then click your browser Reload button.

Agent Specific Information

This section contains information specific to the Agent process that handled the request.

Persistent State Cookies

This section contains a list of all the cookie names received by the browser and the value of each one.

HTTP Headers Sent

This section lists all HTTP headers sent to the browser (for example, Set-Cookie: and Content-Type:). This section will be displayed only if the http option is specified in the URL or on the Administration form.

Miscellaneous Variables

This section contains variables used internally by the WebSpeed Transaction Server, which you can also use in your WebSpeed applications. The SelfURL is always the URL of the current page (minus any arguments), and AppURL is the URL of your application. These variables are derived from the SCRIPT_NAME and PATH_INFO environment variables (see the "Environment Variables" section).

Environment Variables

This section contains all of the environment variables passed to the Messenger process from the Web server. Many of these variables are inherited by the Web server when it is started. Other variables change from request to request.

Form Fields

This section contains fields listed in raw unparsed form and in parsed form that are returned by the get-field( ) API function. This section only contains values if the request is the result of form input or if any arguments are specified following a "?" in the URL.

10.3.5 Utilizing Debugging in Your Application

Any WebSpeed application has access to the WebSpeed global variable debug-options. You can turn on debugging selectively by setting this variable to a comma-separated list of options. As long as your SpeedScript Web object includes `{src/web/method/cgidefs.i}` (this is true by default), you have access to this variable.

The options you can set include "cookie,http,all", among others. The complete list of options resides in *install-path*`/src/web/support/prinval.p`, the WebSpeed procedure that outputs the debugging information for Web pages.

The following code tests if "all" debugging is enabled:

```
IF CAN-DO(debug-options,"all") THEN DO:
  /* add code here */
END.
```

The CAN-DO( ) function compares the second argument with the first, which would be a comma separated list of options If "all" is found in this list, it evaluates to TRUE and executes the SpeedScript in the block.

The following code tests for a custom debugging option called “login” as well as “all”:

```
IF CAN-DO(debug-options, "login") OR CAN-DO(debug-options,"all") THEN DO:
  /* add code here */
END.
```

If you specified debug=login, debug=all, debug=login, oradmin (to name a few), this section of code is executed.

If you use your own debugging options, make sure your application ignores all options it does not understand. Never test the value of the debug-options variable for equality. Always test using the CAN-DO( ) function to see if your custom options are among the listed ones.

10.3.6 Reading the Broker's Log Files

All of the Broker's (and Agents') messages are written to its error and session logs. You can examine the contents of these log files for information about your application. Also, in some cases, messages are written to the database log (.lg) file. For more information about the database log file, see the *Progress Database Administration Guide and Reference*.

Index

Symbols

<!--WSE expression tag 5–29

<!--WSS statement tag 5–28

<%= expression tag 5–29

&DISPLAY 5–13

&OUT 5–13

&OUT-FMT 5–13

&WEBSTREAM 5–13

<% statement tag 5–28

<?WS> statement tag 5–27

` expression tag 5–29

{&FRAME-NAME} 4–21, 4–29

{&OUT}
  compared to DISPLAY output 6–2

{&WEBSTREAM} preprocessor reference 6–2
  setting defaults 6–8

{= expression tag 5–29

A

Adding custom templates 7–7

ADM methods
  HTML-mapping 8–17

Administration 3–11

Agent
  defined 3–2

Agent Variables WebTool 2–10

All-or-nothing processing 9–4

API functions 5–14
  creating 8–13
  date and time 5–22
  detecting configuration 5–24
  general information exchange 5–21
  message handling 5–14
  Web page headers 5–23

AppBuilder 1–3, 2–2
  Code Section Editor 2–4
  connecting to a database server 2–5
  development modes 2–7
  documentation 2–8
  main window 2–5
  object palette 2–2
  running 2–4
  specifying a broker for 2–6

specifying a browser for 2–6
starting 2–5
templates 2–3
wizards 2–3

Application files
appropriate directories 10–3

Architecture 3–1
component interaction 3–5

Arrays
efficient processing 9–14

assign-fields 8–18
data flow 8–22

Attribute list 8–14

Audience xiii

AVAILABLE function 4–11

available-messages API function 5–14

B

Before-image files 9–13

Blank Template 2–3

Blocks
context 5–7
database transactions 9–5
defined 5–5
DO ON ENDKEY 9–7
DO ON ERROR 9–7
DO TRANSACTION 9–7
dynamic resources 5–9
scope 5–8
SpeedScript 5–5
TRANSACTION option 9–5, 9–9
unscoped resources 5–9

Bold typeface
as typographical convention xvi

Broker
defined 3–2
setting up 10–2 to 10–5

browse.html 2–13

building embedded SpeedScript files 5–41

C

CASE statement 4–29

CGI Wrapper
example 4–13
template 2–3
Web objects 4–3
WebSpeed transactions 8–16

check-agent-mode API function 5–24

Code Section Editor 2–4

Command-line utilities 3–11

Comments
SpeedScript 5–4

Commit
two-phase 9–12

Commit unit 9–2

Compile-time code 5–10
interaction with run-time 5–12
syntax 5–10

Compiling Web objects 10–4

Components 3–2
distribution 3–3
interaction 3–5

Configuration 3–11, 5–24

CONNECTED function 9–12

Context
blocks 5–7
defined 5–5

Control handlers
defined 2–12
HTML-mapping 8–17

Controlling WebSpeed transactions 8–4

convert-datetime API function 5–22

Cookies 5–17

Creating
  API functions 8–13
  event procedures 8–13
  method procedures 8–13

Custom tags 5–15, 7–2
  <!--WSMSG --> 5–15

D

Database
  disconnecting 9–12
  integrity 9–2
  servers 2–5
  support 1–2
  WebTool 2–10

Database transactions
  active 9–13
  Agent 9–15
  begin and end 9–5
  blocks 9–5
  committed 8–23
  controlling 9–8
  database 9–2
  database connections 9–12
  defined 9–2
  distributed applications 9–13
  handling 9–2
  improving efficiency 9–14
  input from a file 9–10
  mechanics 9–13
  multi-page 9–15
  multiple databases 9–11
  processing 9–4
  subprocedures 9–10
  subtransactions 9–7
  variables 9–11

Date and time management 5–22

Debugging WebSpeed applications 10–6
  administrative form 10–7
  debug object 10–7
  debug-options variable 10–9
  URL add-ons 10–6
  using cookies 10–6

DEFINE STREAM statement 6–3

Definitions code section
  HTML-mapping 8–17
  SpeedScript-generating 8–16

delete-cookie API function 5–17

Destroy procedure 8–27

Detail Wizard 2–3

Development modes 2–7

Development Server 1–3, 3–4

DISCONNECT statement
  database transactions 9–12

DISPLAY output 6–2
  changing defaults 6–8
  formatting 6–4

displayFields 8–18
  data flow 8–23

Distributed applications
  transaction behavior 9–13

Dynamic HTML pages 5–31

Dynamic resources 5–9

E

Editor WebTool 2–10

Editors 2–4

Embedded SpeedScript 5–25 to 5–42
  authoring files 5–26
  building interactive Web pages 5–30
  conditional execution 5–33
  defining forms 5–32
  defining local data 5–31
  developing Web objects 5–41
  dynamic HTML pages 5–31

examples 4–7
expression escape tags 5–29
form input 4–10
getting input 5–31
HTTP header information 5–40
managing blocks 5–33
<META> and <!--WSMETA--> tags 5–36
output-header coding 5–41
overview 5–25
passing parameters 5–30
running Web objects 5–42
simple query 4–8
statement escape tags 5–27
Web objects 4–2
WebSpeed transactions 8–16
wsoptions 5–37

enable-fields 8–18
effects 8–23

Enterprise Transaction Server 3–4

Environment variables
PROPATH 10–2

Error messages
displaying descriptions xxi

Errors
processing 9–2

Event procedures
assignFields 8–18
calling conventions 5–7
creating 8–13
displayFields 8–18
enableFields 8–18
initialize 8–27
inputFields 8–18
outputFields 8–18
overriding 8–13

Examples 4–5
CGI Wrapper 4–13
directory 2–12
Embedded SpeedScript 4–7
Sports2000 database 4–5
w-cstget.htm 4–16
w-cstget.w 4–19
w-custdir.w 4–31
w-custinf.htm 4–24
w-custinf.w 4–26
w-forcst.w 4–13
w-sstcst.html 4–8
w-sstget.html 4–10

EXCLUSIVE-LOCK 4–30
record phrase 9–5

Executable code, See Run-time code

Expression escape tags 5–29

F

Field objects 4–21
control handlers 8–28

FILE attribute 7–3

File Tools 2–10

Files
before-image 9–13
local-before-image 9–14

FIND Records
HTML-mapping
data flow 8–23

FIND statement 4–30

Form
buffer 4–20
hidden fields 5–19
input 4–10

format-datetime API function 5–22

Formatting DISPLAY output 6–4

Frame buffer 8–22

Frameset Template 2–3

frameset.html 2–13

Function API
defined 2–11

Functions
CONNECTED 9–12
TRANSACTION 9–13

G

get-config API function 5–24

general information exchange 5–21

GET request method 8–20

get-cgi API function 5–21

get-cookie API function 5–17

get-field API function 5–21

get-message-groups API function 5–14

get-messages API function 5–14

get-transaction-state procedure 9–15

get-user-field API function 5–20

get-value API function 5–21
access to form buffer 4–22
called from input-fields 4–22
example 4–29

get-value function 4–11

GLOBAL-DEFINE 6–2

H

Handling messages 5–14

Handling time outs 8–33

headers
Web page 5–23

Help
WebSpeed messages xxi

Help WebTool 2–10

Hidden form fields 5–19

hidden-field API function 5–19

hidden-field-list API function 5–19

hmapmain.i 2–13

HTML files
deploying 10–3

HTML format attributes 6–6
setting defaults 6–8

HTML mapped Web objects 4–3

HTML Mapping
complex 4–22
examples 4–15
simple HTML Mapping 4–16
with a SmartDataObject 4–31
wizard 2–3

HTML mapping
transactions 8–17

HTML pages
mapped to Web objects 4–20

HTML table prototype 7–2

HTML visualizations 7–2

html-encode API function 5–21

HTML-END-OF-LINE 6–6

HTML-END-OF-PAGE 6–7

HTML-FRAME-BEGIN 6–6

HTML-FRAME-END 6–6

HTML-HEADER-BEGIN 6–6

HTML-HEADER-END 6–7

html-map.w 2–13

HTML-mapping Web objects
creating definitions 8–25
moving data 8–21

HTML-TITLE-BEGIN 6–7

HTML-TITLE-END 6–7

htm-offsets code section
  HTML-mapping 8–17

HTTP headers 4–25
  embedded SpeedScript 5–40

I

IF...THEN...ELSE function 4–11

include.i 2–13

Information
  exchange 5–21
  passing 5–16

input-fields 8–18
  data flow 8–22

INTEGER function 4–21

Internal procedures
  defining 5–5
  running 5–6

Italic typeface
  as typographical convention xvi

Iteration 9–5

J

Java class files
  deploying 10–3

K

Keystrokes xvi

L

Licensing options 2–2

Load balancing 3–4

Local before-image files 9–14

Local development mode 2–7

Locking records
  handling 9–5

Log files 10–10

Looping 9–5

M

Main Code Block section
  HTML-mapping 8–17
  modifying 8–26
  SpeedScript-generating 8–16

Main Template 2–3

main.html 2–13

Management 3–11

Manual
  organization of xiv
  syntax notation xvii

Message handling 5–14

Messages
  displaying descriptions xxi

Messages WebTool 2–10

Messenger
  defined 3–2

<META> tags 5–36

method directory 2–12

Method procedures
  creating 8–13
  defined 2–11
  output-header 8–11
  process-web-request 4–14, 4–30, 8–13, 8–18
  running 5–6

Monospaced typeface

as typographical convention xvi

Multi-database applications 9–11

Multi-page database transactions 9–15
example Web object 9–16
flow chart of model 9–19
guidelines 9–18

N

NAME attribute 7–3

NameServer
defined 3–2

NO-ERROR option 4–11

Nonexecutable code, see Compile-time code

Nonvisual objects 2–2

NO-UNDO option 4–27
variables defined 9–11, 9–14

NSCONFIG 3–11

NSMAN 3–11

O

Object palette 2–2

Object State WebTool 2–10

objects directory 2–13

Offsets
deploying 10–4
files 4–17
generating 8–31

OS Command WebTool 2–10

{&OUT} 4–15

Output
directing to the Web 6–2

output stream 5–13

OUTPUT TO "WEB" statement 6–3
formatting options 6–4

output-content-type API function 5–23

outputFields 8–18
data flow 8–23

output-header
coding in embedded SpeedScript 5–41
customizing 8–11

output-http-header API function 5–23

output-messages API function 5–14

Overriding event procedures 8–13

P

PAGED option 6–4

PAGE-SIZE option 6–4

Passing information 5–16

Persistence
Web object 8–10

Persistent procedures 5–6

POST request method 4–30, 8–20 to 8–22

Preprocessors 5–13

Primary Web objects 8–8

PROADSV 3–11

procedur.p 2–13

Procedures
persistent 5–6
running 5–6

Processing
all-or-nothing 9–4

process-web-request 8–18, 8–27

customizing 8–13
editing 4–18
mapping anatomy 8–18
standard HTML mapping 8–19

Progress Explorer 3–11

PROPATH 4–6

PROPATH environment variable 10–2

ProPath WebTool 2–10

ProVision AppBuilder 2–2

Provision Plus AppBuilder 2–2

Q

query strings 5–18, 5–35

QUERY_STRING variable 8–20

queue-message API function 5–14

R

Record buffer 8–22

Records
locks 9–5

Remote development mode 2–7

Report Template 2–3

Report Wizard 2–3

Request logic 8–20

REQUEST_METHOD variable 8–20

Retrieving general information 5–21

RUN statement
PERSISTENT option 5–6

Running procedures 5–6

Run-time code 5–11
interaction with compile-time 5–12
object names 5–12
syntax elements 5–11

S

Sample applications 2–10

Sample databases
defined 2–12

Sample Web objects 4–1

Scope 5–8
block context 5–8
dynamic resources 5–9
unscoped resources 5–9

SCREEN-VALUE 4–21, 8–32

<SCRIPT LANGUAGE> statement tag 5–27

script.html 2–13

Scripting Lab 2–10

Secondary Web objects 8–8

Security
application 10–5

Semantics of SpeedScript
context 5–7

SENSITIVE attribute 8–23, 8–24

<SERVER> statement tag 5–27

set-cookie API function 5–17

set-transaction-state procedure 9–15

set-user-field API function 5–20

SKIP option 4–15

SmartDataObject 2–2, 4–31

Source files 2–12

SpeedScript
AVAILABLE function 4–30
blocks 5–5
comments 5–4
compared with 4GL 5–2
compile-time compared to run-time 5–10
defined 1–5, 2–11
documentation 1–5
embedding in HTML 5–25, 5–27
form buffer 4–20
generating messages 5–15
INTEGER function 4–21
objects 5–6
semantics
dynamic resources 5–9
scope 5–8
statements 5–4
syntax 5–4
triggers 5–7
WebSpeed preprocessors 5–13
XML support 5–24

SpeedStart 3–11

Sports2000 database 2–12, 4–5

Standard Web objects 4–2

Starting databases and Brokers 10–5

State
defining 8–2
Web object 8–2
WebSpeed Application 8–3

State-aware
default time-out handling 8–36
resetting time-outs 8–35
time outs 8–33
time-out handler 8–34
Web objects 8–5

Stateless applications 8–3

statement escape tags 5–27

Statements
executable 5–10
nonexecutable 5–10
SpeedScript 5–4

State-passing applications 8–3

State-persistent applications 8–3
advantages and drawbacks 8–3

State-persistent Web objects 8–7

static Web objects 4–2

stream 5–13

Subprocedures
database transactions 9–10

Subtransactions 9–7
mechanics 9–14

Super procedures 2–14

support directory 2–13

Syntax
checking 4–21
SpeedScript 5–4

Syntax notation xvii

T

Table template 2–3, 7–5

table.html 2–13

TagExtract 8–31
pathnames 8–31

tagmap.dat file 8–22, 8–28, 8–29
defined 2–12
deploying 10–4
utility procedures 8–29

template directory 2–13, 2–14

Templates 2–3, 2–13
adding custom 7–7

time management 5–22

tools 2–1

Transaction Broker

default directory
creating 10–2

Transaction Control 8–11
with CGI Wrapper Web objects 8–16
with embedded SpeedScript Web objects 8–16
with HTML mapping Web objects 8–17

TRANSACTION function 9–13

TRANSACTION option
blocks 9–5, 9–9

Transaction Server 3–4

Transaction blocks 9–5

Triggers 5–7

Two-phase commit 9–12

Typographical conventions xvi

U

UNDO processing 9–3, 9–11

Unscoped resources 5–9

url-decode API function 5–18

url-encode API function 5–18

url-field API function 5–18

url-field-list API function 5–18

url-format API function 5–18

URLs
building query strings 5–35
query strings 5–18
Web objects 4–6
WebSpeed Broker 2–6

User fields 5–20
list 8–15

V

VALUE option
run-time object names 5–12

Variables
database transactions 9–11

Virtual System Table WebTool 2–10

Visual objects 2–2

Visualizations
HTML-mapping 7–2
other than HTML 7–8
SpeedScript-generated 7–6

W

w-cstget.htm 4–16

w-cstget.off 4–17

w-cstget.w 4–16, 4–19

w-custdir.w 4–31

w-custinf.htm 4–24

w-custinf.w 4–22

Web browser 1–3

Web objects 4–1, 4–2
basic types 4–2
calling conventions 5–6
CGI Wrapper 4–3
compiling 10–4
criteria for using 4–4
data flow 8–21
deploying 10–3
examples 4–5
HTML mapped 4–3
input 8–21
making secure 10–5
mapped to HTML pages 4–20
primary 8–8
running 5–6
secondary 8–8

standard 4–2
state 8–2
state context 8–7
state-aware 4–29, 8–2, 8–5
state-aware and persistence 8–10
stateless 8–2
transaction control 8–11
URLs 4–6

Web output stream 5–13

Web page headers 5–23

web.cst 2–13, 7–7

web.input 8–22, 8–32
defined 2–12

web.output 8–23, 8–29, 8–32
defined 2–12

WEB-CONTEXT handle
HTML attributes 6–6

web-disp.p
defined 2–12

WebSpeed
administration 3–11
advantages 1–2
Agent 3–2, 3–4, 3–5
API functions 5–14
AppBuilder 2–2
application states 8–3
applications
starting and stopping 10–5
architecture 3–1
Broker 2–6, 3–2
command-line utilities 3–11
components 1–3, 2–11, 3–2, 3–3, 3–5
database support 1–2
debugging applications 10–6
defined 1–2
deployment environment 1–3
development environment 1–3
development server 1–3
examples 4–5
examples directory 2–12
global variables 2–11
handling requests 3–5
Messenger 3–2
method directory 2–12
objects directory 2–13
output stream 6–2
overview 1–3
preprocessor definitions 2–11
preprocessors 5–13
sample Web objects 4–1
source files 2–12
SpeedScript 1–5
super procedures 2–14
support directory 2–13
template directory 2–13, 2–14
templates 2–3
tools 2–1
Transaction Server 3–4
transactions 8–4
wizards 2–3
Workshop 1–3

Web-Timout-Handler 8–34

WebTools 1–3, 2–9
documentation 2–10

w-forcst.w 4–13

wizards 2–3

Working application directory
creating 10–2

wrap-cgi.w 2–13

WSCONFIG 3–11

<!--WSMETA --> tags 5–36

<!--WSMSG --> custom tag 5–15

wsoptions 5–37

w-sstcst.html 4–8

w-sstget.html 4–10

WTBMAN 3–11

X

XML 5–24