Progress®

OpenEdge Application Server: Developing WebSpeed Applications

21 June 2023

Copyright

Table of Contents

1

Developing with WebSpeed

This chapter is an overview on Web programming with WebSpeed on the OpenEdge platform.

Related Links

- Introduction to WebSpeed
- WebSpeed and the OpenEdge platform
- Web programming and WebSpeed

Introduction to WebSpeed

Progress® WebSpeed® is a Progress® ABL development and deployment environment. WebSpeed allows you to build applications that use HTML, XML, WML, DHTML, and most other mark-up languages (MLs) as the user interface. This means that WebSpeed can be used for applications where users are accessing the application using a Web browser, a mobile/cell-phone, or some other system making requests for information using XML and HTTP or HTTP/S as the transport protocol.

In addition to building Web-based applications by design, you can use WebSpeed to Web-enable existing Progress® OpenEdge® applications that previously ran stand-alone or as an Progress® AppServer-based application.

With WebSpeed, you can develop and deploy:

- Intranet applications that allow internal users to access and modify data.
- Internet applications that allow external, consumer access (for example, a shopping cart application).
- Extranet, business-to-business applications.

Note:

This documentation supports the "classic" WebSpeed implementation, where a third party Web server is required to host the WebSpeed Messenger that responds to client requests.

In OpenEdge 11.6 and later releases, the Progress Application Server for OpenEdge (PAS for OpenEdge) is available as a host for WebSpeed applications. It integrates the functionality of a Web server, the WebSpeed Transaction server, and an AppServer in a single PAS for OpenEdge instance. As a platform for WebSpeed applications, PAS for OpenEdge has many advantages, including improvements in performance, scalability, and security.

Note that you can migrate existing WebSpeed applications to PAS for OpenEdge, or you can use the Progress Developer Studio for OpenEdge to develop new WebSpeed applications that run on PAS for OpenEdge.

See *Progress Application Server for OpenEdge: PAS for OpenEdge* for more information on the advantages of using PAS for OpenEdge.

WebSpeed and the OpenEdge platform

As a part of OpenEdge, WebSpeed applications can connect to the rest of the OpenEdge application server platform. For instance, a WebSpeed application can call other OpenEdge applications across an AppServer. You should become familiar with the other parts of the OpenEdge platform, as well as WebSpeed. For a basic look at all the pieces of the application server picture, see *OpenEdge Getting Started: Application and Integration Services*.

You should also look at the information in *OpenEdge Getting Started: WebSpeed Essentials*. That manual describes the WebSpeed architecture and how it operates; the WebSpeed development tools; distributing, configuring, and securing WebSpeed environments; and running WebSpeed applications.

At their core, WebSpeed applications are Progress ABL applications. Generally speaking, most things you can do with the Progress ABL, you can do with WebSpeed. However, coding your entire application as a WebSpeed application is not necessarily the best practice. When you plan to deploy an application on multiple clients, you should modularize the code so that only tasks that differ between clients are duplicated. Business logic that is not client-specific should be shared by all clients. The more modular your code is, the easier it is for you to maintain and reuse the code. Modularization is just one practice that makes your initial development efficient and eases later efforts to adapt to changing technology and business needs.

Related Links

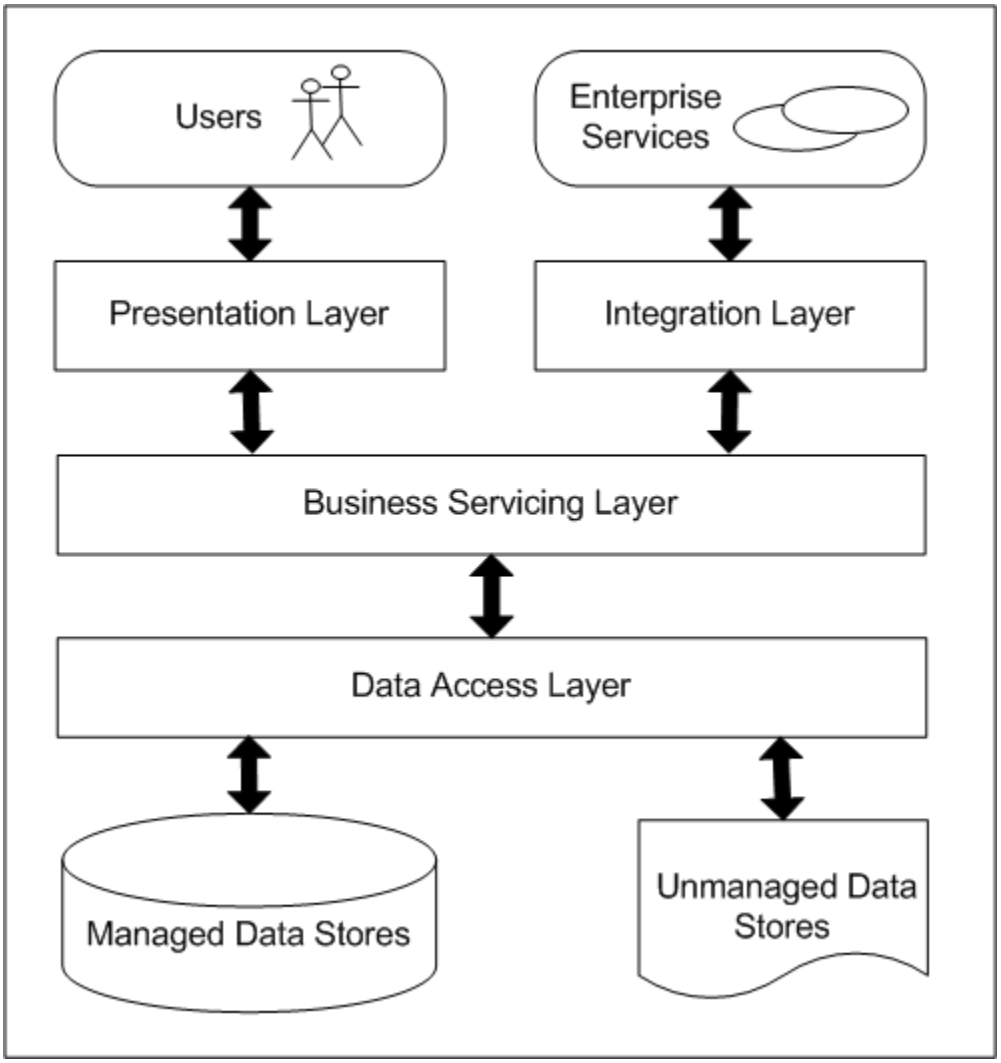
- The OpenEdge Reference Architecture
- WebSpeed and the OpenEdge Reference Architecture

The OpenEdge Reference Architecture

The OpenEdge Reference Architecture is a recommended approach to designing business applications according to current best practices. The reference architecture views an application as a set of layers which provide services to each other, as shown in the following figure. This model allows your business and support

logic to be modularized for flexibility and reusability. You can use the reference architecture as a whole, or adopt it a piece at a time to fit your needs. the reference architecture separates business tasks into a set of layers.

Figure 1. OpenEdge Reference Architecture



The procedures in the Data Access layer manage handling data from your data stores. These procedures retrieve information from wherever it resides in the physical data stores and arrange the data into logical datasets that meet the business needs for the procedures in the Business Servicing layer. Other procedures in the Data Access layer extract the data changes from the logical datasets and commit the changes to the proper places in the physical data stores.

The procedures in the Business Servicing layer act on requests received from users through the Presentation layer or from enterprise services through the Integration layer. These procedures handle the business tasks required to fulfill an order, for example. The procedures in the Business Servicing also push data changes in the data sets back to the Data Access layer.

The procedures in the Presentation and Integration layers pass requests from external sources (users or enterprise services) to the Business Servicing layer. Procedures in these layers might prevalidate that user requests are complete and in the proper format. The main work of these layers is to transform incoming data into the form needed by the business logical of the Business Servicing layer and to properly present the results for the consumers.

This is only a brief sketch of the OE Reference Architecture. For more information, see OERA Wiki on Progress Communities Web site.

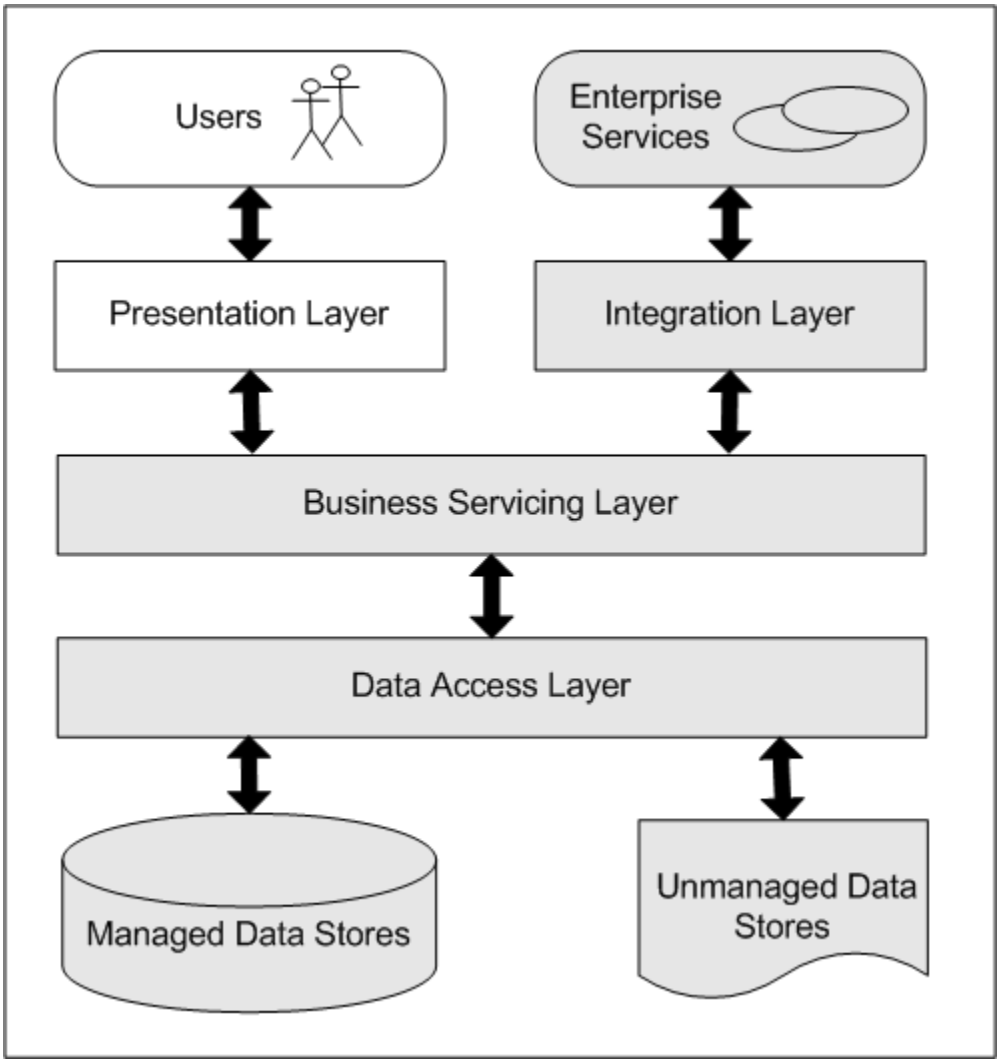
WebSpeed and the OpenEdge Reference Architecture

If you adopt the OE Reference Architecture, you need to consider where WebSpeed applications are appropriate choices. Before coding a task into the WebSpeed part of your application, you should consider if it really belongs there. What is the essential difference between a WebSpeed application and an equivalent Progress ABL application?

A WebSpeed application and an equivalent Progress ABL application use the same data to complete the same business task. So, they would use the same modules in the Data Access layer. To perform the same task, both applications would use the same business logic. So, they would use the same modules in the Business Servicing layer.

The essential difference between WebSpeed and Progress ABL applications is how they gather information from and present results to the user. A WebSpeed application uses an HTML client (or a client based on some other markup language). This point positions WebSpeed applications as elements of the OE Reference Architecture's Presentation layer as shown in the following figure.

Figure 1. WebSpeed's use in OpenEdge Reference Architecture



In an application built according to the reference architecture, a WebSpeed component passes user requests to the appropriate procedures in the Business Servicing layer and passes the results back to the user. This role limits the kinds of tasks that you would code into WebSpeed procedures.

For example, the following tasks are generally appropriate for the Presentation layer:

- Validating that the fields in a form are filled in with appropriate values
- User interface control tasks, such as populating a secondary combo box based on the selection in the primary combo box

The following tasks are generally not appropriate for the Presentation layer:

- Calculation routines, such as figuring price totals or sales tax

- Direct database access for anything other than a UI control task

Web programming and WebSpeed

WebSpeed is used to manipulate, customize, and automate facilities for web-based applications. It allows you to develop and deploy Internet-based applications that use XML, HTML, DHTML, WML, and Java by embedding Progress® SpeedScript® directly into your HTML pages, or using HTML mapping to bind HTML files to business logic.

WebSpeed can be deployed in environments leveraging:

- Cascading Style Sheets (CSS) — Provides a simple mechanism for adding style characteristics to Web documents. For more information, refer to www.w3.org/Style/CSS.
- Extensible Markup Language (XML) — XML is a simple and flexible text format derived from SGML. It was originally designed to meet the challenges of large scale electronic publishing, but it is also playing an important role in the exchange of a wide variety of data on the Web. For more information, refer to www.w3.org/XML.
- Wireless Markup Language (WML) — WML inherits traits based on HTML and XML and is used to run simple code on the client.
- Hypertext Markup Language (HTML) — HTML is the standard language for publishing hypertext on the Web. It is a non-proprietary format based on SGML, and can be used to process a wide range of tools.
- Dynamic Hypertext Markup Language (DHTML) — DHTML allows you to control the display and positioning of HTML elements in the browser. This language is a combination of HTML, CSS, and JavaScript.

Web-based applications developed using WebSpeed are run in a web browser. A web browser provides the host environment of client-side computation, including objects representing windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies and input/output functionality. In addition, the web browser provides a means to attach scripting code to events such as a change of focus, page and image loading, unloading, error and abort, selection, form submission and mouse actions. WebSpeed coding appears within the HTML, and the displayed page is a combination of user interface elements and fixed and computed text and images.

2

Web Objects

This chapter describes the three types of Web object that you can build with WebSpeed. The sample Web objects that are included with WebSpeed are used as examples and illustrate some basic WebSpeed techniques and concepts.

Related Links

- Types of Web object
- Using Web object examples
- Embedded SpeedScript examples
- A CGI Wrapper example
- HTML mapping examples

Types of Web object

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

Related Links

- Standard or static HTML Web objects
- Embedded SpeedScript Web objects
- CGI Wrapper Web objects
- HTML Mapped Web objects

- General criteria for use

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 regenerated 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.

Embedded SpeedScript Web objects

SpeedScript is a subset of the Progress ABL 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 Embedded SpeedScript examples. Also see Overview of Embedded SpeedScript more information about embedded SpeedScript.

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 A CGI Wrapper example for more information.

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 Progress® 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 HTML mapping examples for more information.

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 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 sections that follow 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.

Using Web object examples

Before you can run examples, you must have a WebSpeed development environment installed and configured. See *OpenEdge Getting Started: WebSpeed Essentials* for information about setting up WebSpeed.

To view or run the example Web objects:

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 the AppBuilder.
3. Choose **Tools > Database Connections** and connect to a working copy of the Sports2000 database.

   The Sports2000 database is a sample OpenEdge 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.

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 File Tools, 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 using the utilities in File Tools.

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 *OpenEdge Getting Started: WebSpeed Essentials* for more information about WebSpeed sample applications.

Related Links

- Web object URLs

Web object URLs

You use standard URL notation to access Web objects from a browser.

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 Progress® WebSpeed® 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. 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 *OpenEdge Application Server: Administration*.

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 Using Web object examples.

Related Links

- A simple query
- Handling form input

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 FIELDS(CustNum Name Phone):  </script>
  <tr>
    <td align=left> 'Customer.CustNum' </td>
    <td align=left> 'Customer.Name' </td>
    <td align=left> 'Customer.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.

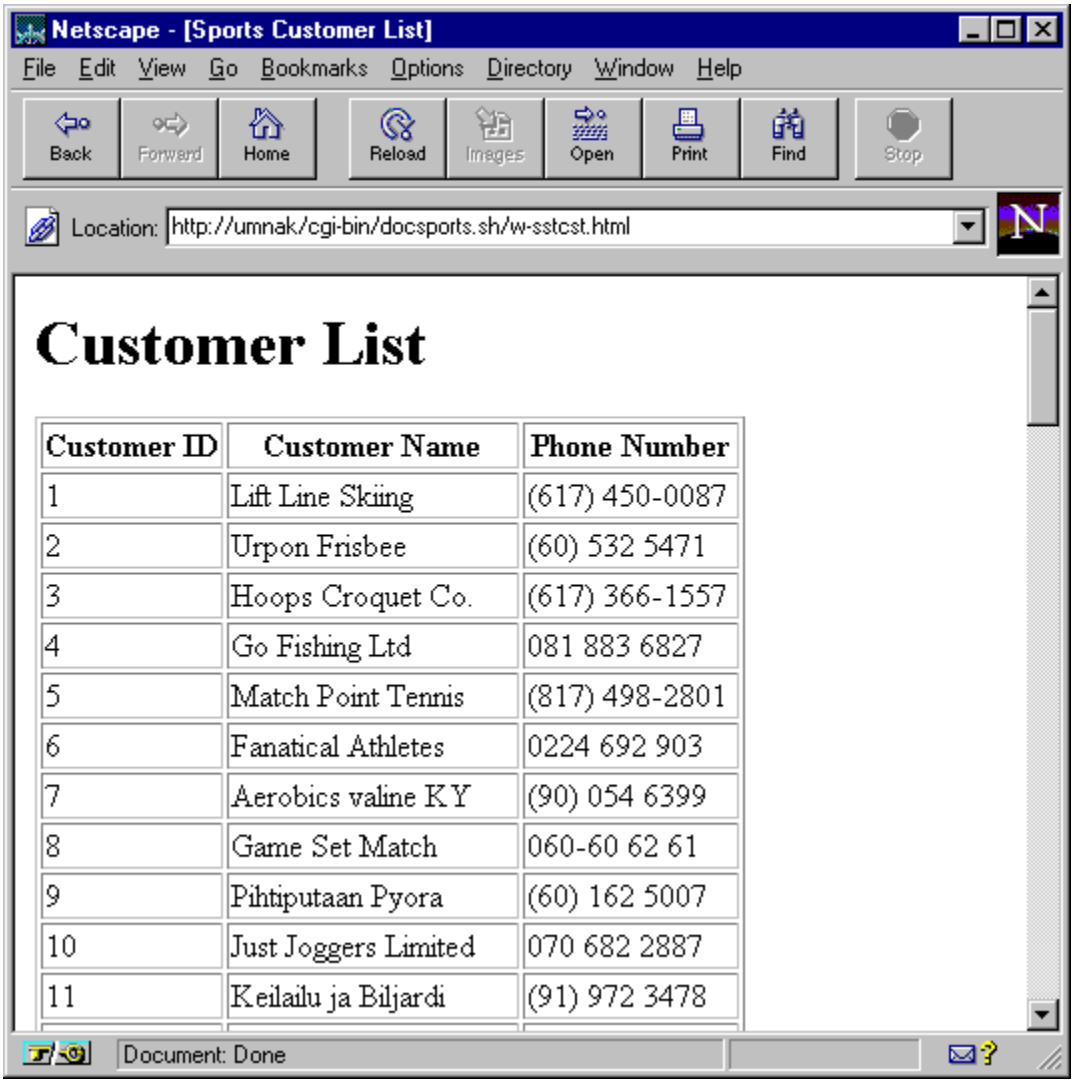
- Expression escape tags (`` `...` ``), which allow you to include the current value of a single data element directly into your HTML source.

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.

The following shows the output when you compile and run w-sstcst.html.

Figure 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 Statement escapes and Expression escapes.

Handling form input

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

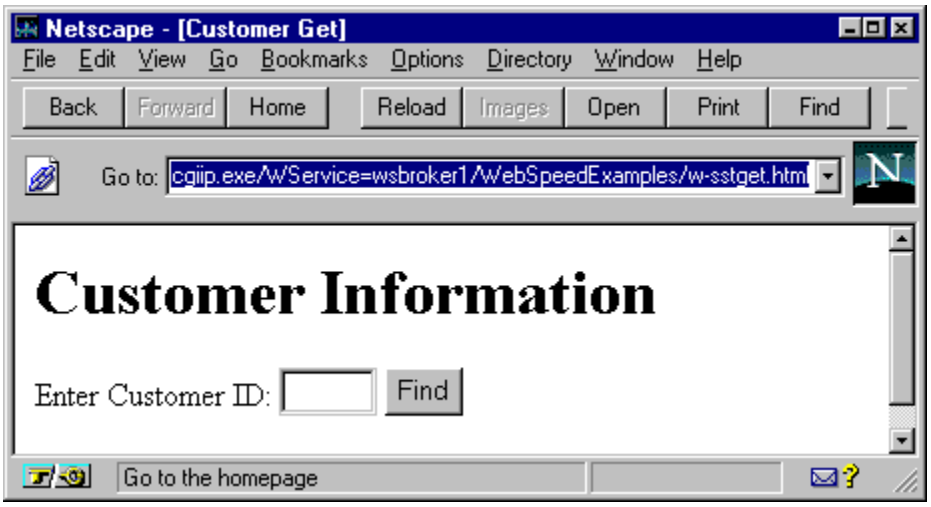
w-sstget.html

```
. . .
<h1>Customer Information</h1>
<!--WSS
IF get-value("CustNum") NE "" THEN
  FIND Customer
    WHERE Customer.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(Customer.CustNum) ELSE ""`/>
    <input type="SUBMIT" name="Button" value="Find"/>
  <br/><br/>
  <!--WSS
  IF AVAILABLE(Customer) THEN DO: -->
  Name: <input name="Name" size="20" value=" `Customer.Name` "/>
  --------Phone: <input name="Phone" size="20" value=" `Customer.Phone` "/>
  <br/><br/>
  Address 1: <input name="Address" size="20" value=" `Customer.Address` "/>
  <br/>
  Address 2: <input name="Address2" size="20" value=" `Customer.Address2` "/>
  <br/><br/>
  City: <input name="City" size="12" value=" `Customer.City` "/>
  ---State: <input name="State" size="20" value=" `Customer.State` "/>
  ---Zip: <input name="PostalCode" size="10"
    value=" `Customer.PostalCode` "/>
  <br/><br/>
  Comments: <textarea name="Comments" rows="2" cols="30">
  `Customer.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 the following figure.

Figure 1. 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 WebSpeed API Reference. You can also access the WebSpeed API Reference from 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 for an overview of some commonly used API functions.

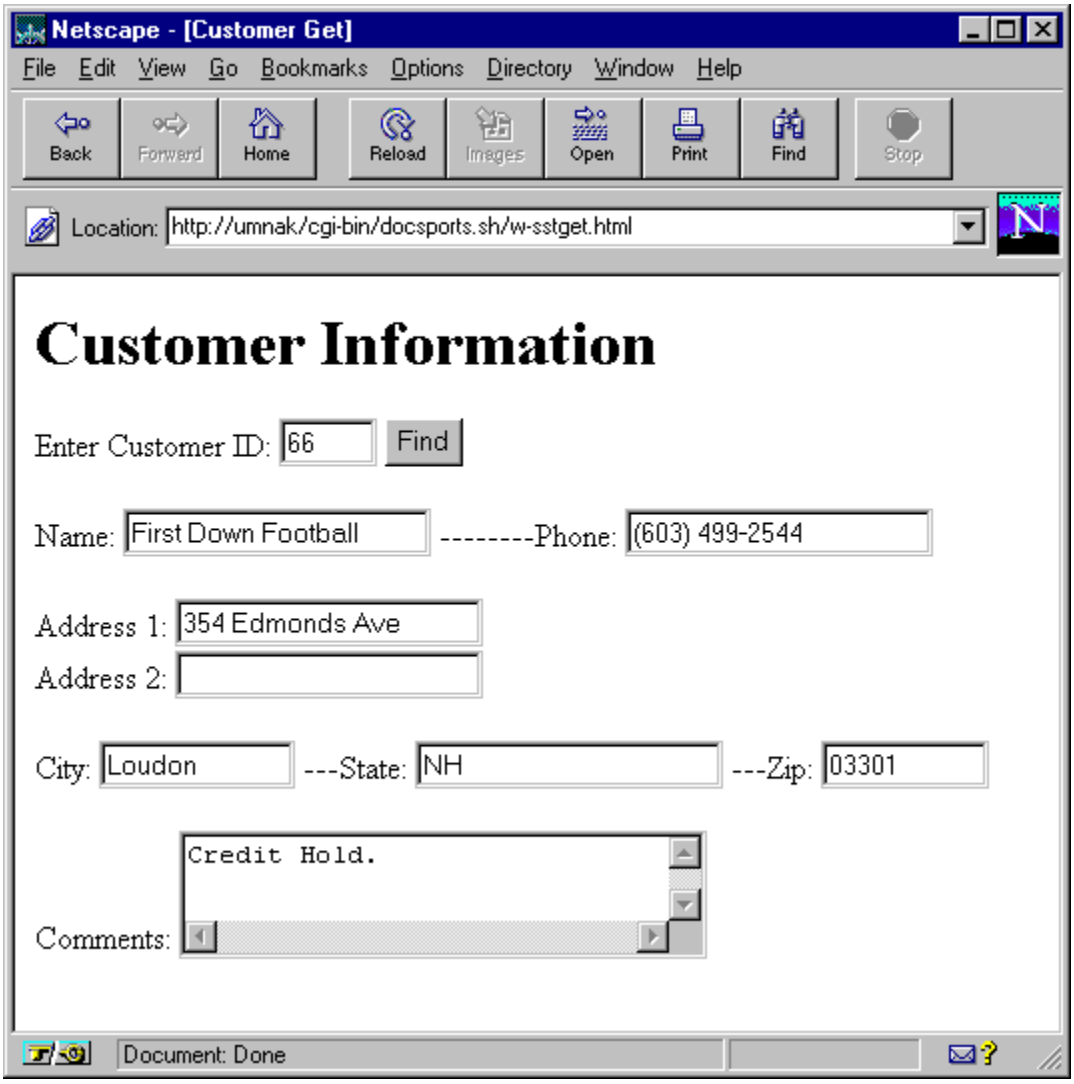
Note: The NO-ERROR option on the FIND statement allows the Web object to continue execution if the FIND statement fails because the CustNum value is invalid.

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.

The following figure shows the HTML output as it would appear in a browser.

Figure 2. 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.

A 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 FIELDS(CustNum Name Phone) NO-LOCK:
      {&OUT} "<TR>":U SKIP
             "    <TD ALIGN=LEFT>":U Customer.CustNum "</TD>":U SKIP
             "    <TD ALIGN=LEFT>":U Customer.Name "</TD>":U SKIP
             "    <TD ALIGN=LEFT>":U Customer.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 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 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 A simple query.

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 *OpenEdge Development: ABL Reference*.

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 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.

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 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.

Related Links

- Simple HTML mapping
- Complex HTML mapping
- Complex HTML mapping that includes a SmartDataObject

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.

Related Links

- Mapping form elements to database fields
- Generating the offset file
- Editing process-web-request
- Compiling and running
- SpeedScript form buffer
- An alternative to form buffer input

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"
```

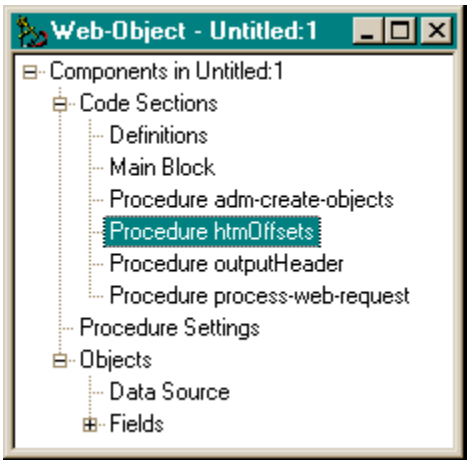
```
field[5]= "Address2,INPUT,,fill-in,16,12,16,44"
field[6]= "City,INPUT,,fill-in,17,7,17,35"
field[7]= "State,INPUT,,fill-in,18,11,18,40"
field[8]= "PostalCode,INPUT,,fill-in,19,9,19,43"
field[9]= "Comments,TEXTAREA,,editor,20,11,20,66"
```

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 should generate a new offset file. All of positional information might become invalid even if you make minor changes to your source files. Since manual editing is error prone, you should use the HTML Mapping Wizard and to generate a new offset file. You can also automatically re-map by opening the Web object in the AppBuilder. An updated offset file will be generated. The Progress® 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 the following figure.

Figure 1. 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 the `process-web-request` procedure (listed under **Code Sections** in the tree view of the HTML mapping source) in the Section Editor, 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 *OpenEdge Development: ABL Reference*.

Also note that `process-web-request` runs a different sequence of processes depending on whether the `METHOD` attribute for the HTML form element 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}).
  . . .
```

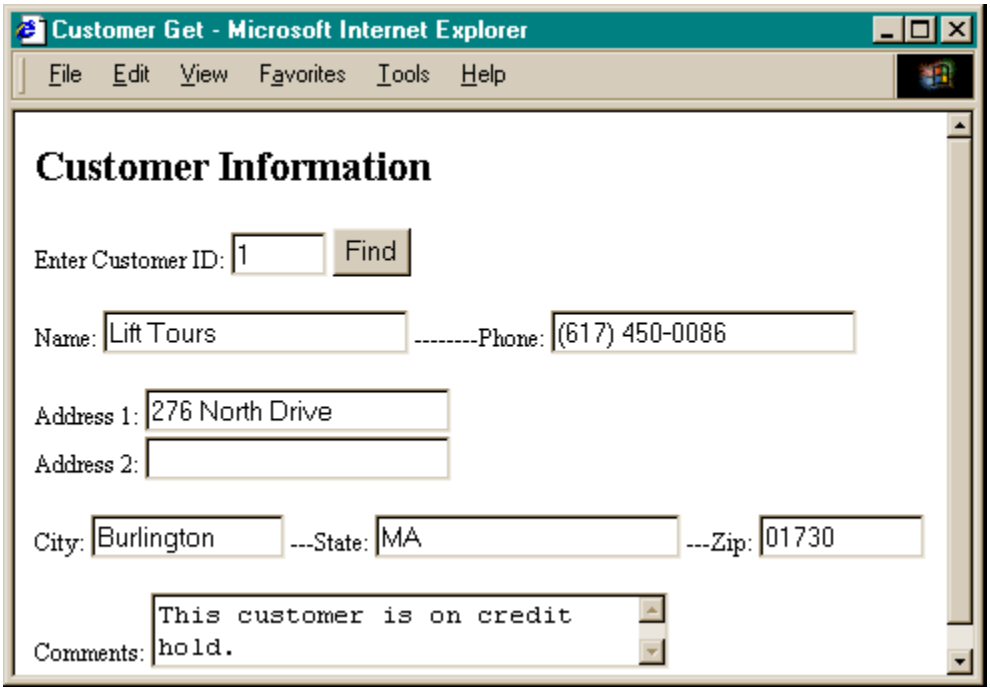
Notice that the default `Run findRecords` was commented-out.

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 the following figure.

Figure 1. 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 ABL 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 ABL documentation while doing WebSpeed development. For more information on how input and output differs between the Progress ABL and SpeedScript, see SpeedScript and Progress ABL.

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 see Customizing field object control handlers.

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. It 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.
```

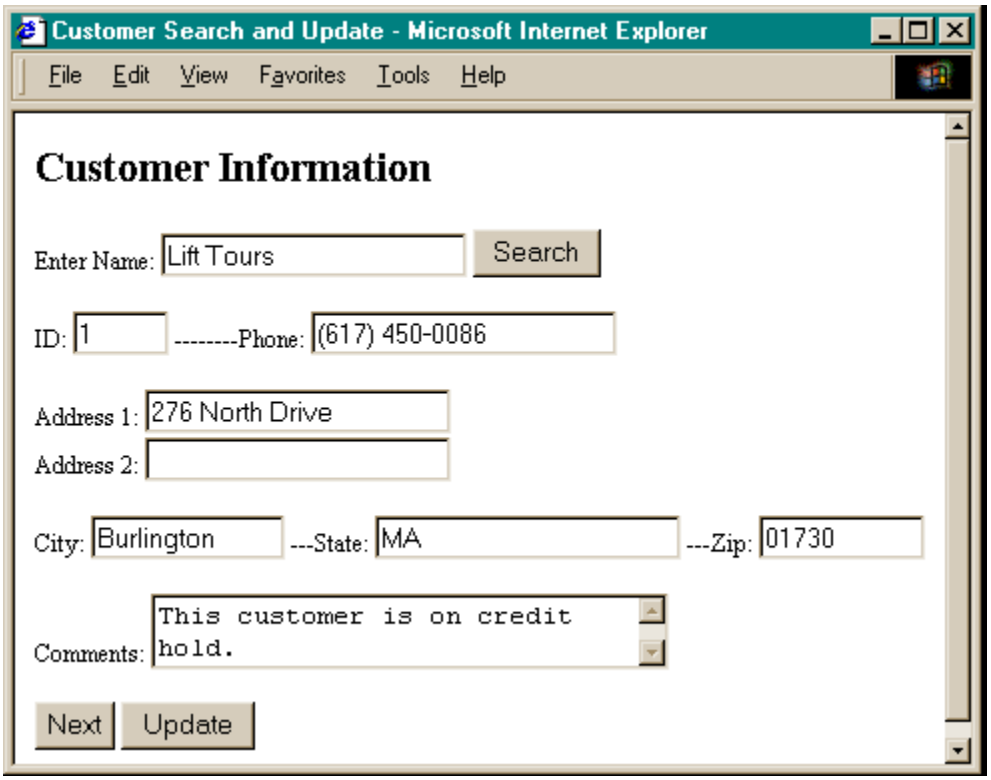
As described previously for embedded SpeedScript Web objects, WebSpeed provides the get-value API function to return values from a Web page. For the CustNum form element, this function accesses the value directly from HTML input. Using this technique avoids any need to reference the form buffer to retrieve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

Complex HTML mapping

The following figure 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 1. Web page generated from w-cstinf.w



Basically, the Web object does the following:

- Initially, it generates a Web page similar to the one shown above. 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 following 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 the Web page shown in Figure 1. The Web page is interactive. The user can view a different record by using the **Submit** or **Nex**t 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 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

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, 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

---

For "Update", you must find a unique record that corresponds to the returned values, because there can be duplicate customer names. Since 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. If there is no such record, or the specified record is locked by another 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 new line in a string.

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

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.
```

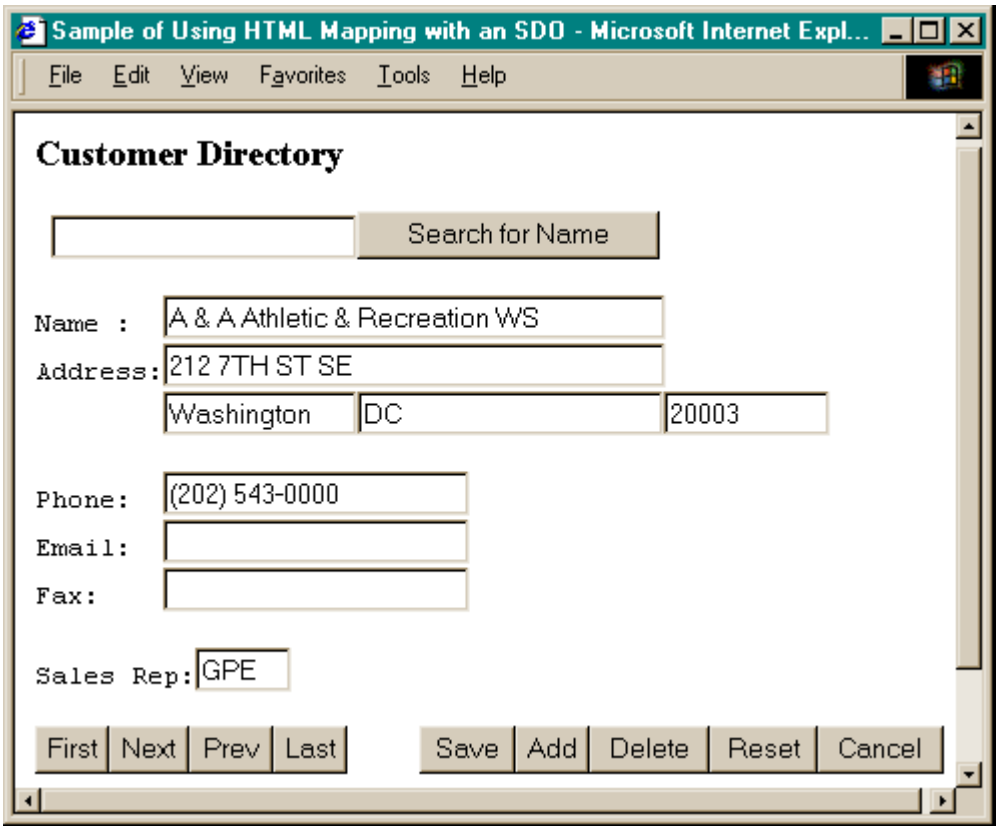
The test of `vButton` is added, again, because the next default event procedure, `assignFields`, assumes that you want to update the current record with the values received from the user. However, this is only true if the user submitted the form for update, and only if the specified record was actually 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.

Complex HTML mapping that includes a SmartDataObject

The following figure shows the Web page generated by another sample WebSpeed application, w-custdir.w.

Figure 1. Web page generated from w-custdir.w



This Web page 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.

Related Links

- Creating a SmartDataObject
- Mapping with a SmartDataObject

- Adding code

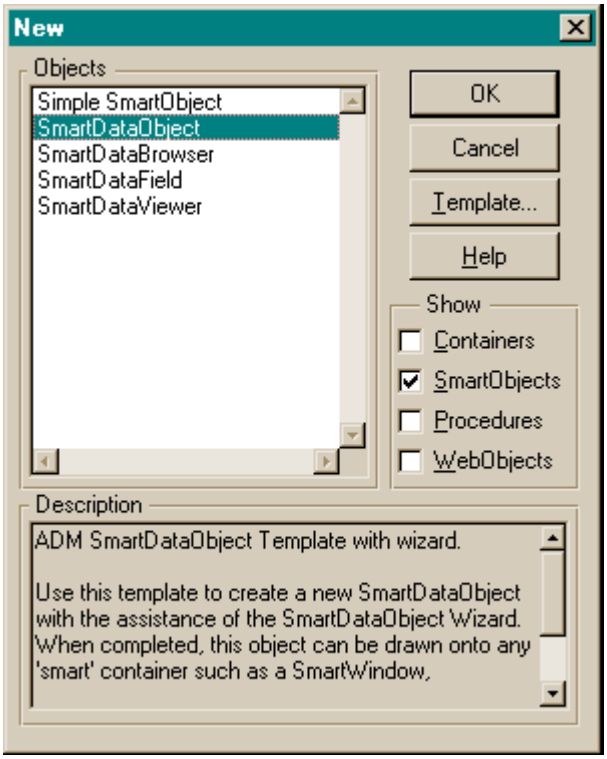
Creating a SmartDataObject

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 *OpenEdge Application Server: Developing AppServer Applications* for more information.

SmartDataObjects can easily be created in the AppBuilder. The SmartDataObject associated with w-custdir.w is called dcustomer.w.

To create SmartDataObjects in the AppBuilder:

1. Choose **File > New** from the AppBuilder main menu. The **New** 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 *OpenEdge Getting Started: WebSpeed Essentials*.

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 *OpenEdge® Development: ADM and SmartObjects®* and *OpenEdge® Development: ADM Reference*.

For more information about creating SmartDataObjects with the AppBuilder, see the *OpenEdge® Development: AppBuilder*.

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>
```

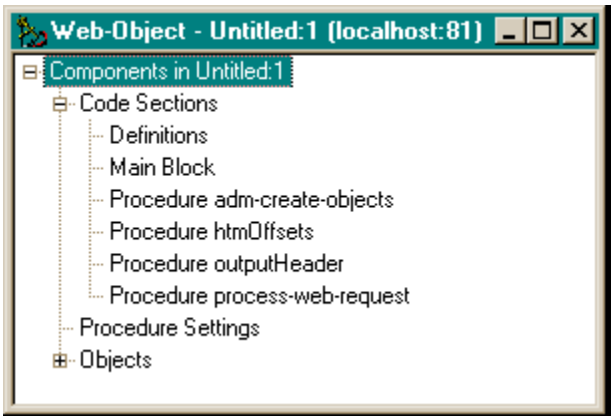
You can use the HTML Mapping Wizard's Automap feature to map the fields referenced in w-custdir.htm to the database fields referenced in dcustomer.w. Note that all the fields will map except the hidden fields and the `SearchName` field (shown in bold). These fields are used to pass information to the 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 added 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 the following figure.

Figure 1. 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 *OpenEdge Development: AppBuilder*.

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 *OpenEdge Development: ADM and SmartObjects* and *OpenEdge Development: ADM 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.
    UN 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 () NE "add" THEN
        RUN fetchCurrent.
      RUN assignFields.
      setAddMode (FALSE).
      setUpdateMode ("").
    END.
    WHEN "Delete" THEN DO:
      RUN fetchCurrent.
      deleteRow().
    END.
    WHEN "Reset" THEN
      RUN fetchCurrent.
    WHEN "Cancel" THEN DO:
      RUN fetchCurrent.
      setUpdateMode ("").
    END.
    WHEN "Add" THEN DO:
      RUN fetchCurrent.
      setUpdateMode ("Add").
    END.
  END CASE.
```

3

SpeedScript

This chapter contains information about SpeedScript and embedded SpeedScript.

Related Links

- SpeedScript and Progress ABL
- Elements of SpeedScript syntax
- WebSpeed preprocessors
- WebSpeed API functions
- XML support in SpeedScript
- Overview of Embedded SpeedScript
- Authoring embedded SpeedScript files
- How to embed SpeedScript in HTML
- Building interactive Web pages with embedded SpeedScript
- Using <META> and <!--WSMETA --> tags
- Compiling and running embedded SpeedScript Web objects

SpeedScript and Progress ABL

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

- *OpenEdge Development: ABL Reference* — Describes the syntax of all SpeedScript language elements. Also identifies which language elements apply to ABL, to SpeedScript, or to both ABL and SpeedScript.
- *OpenEdge Getting Started: ABL Essentials* — Covers important programming concepts such as database locking and transaction rules, program block and resource scoping rules, and the use of persistent procedures.

Related Links

- Similarities
- Differences

Similarities

The similarities between ABL 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).

Differences

Some important differences between SpeedScript programming and Progress ABL programming are:

- ABL applications are usually state-aware, while SpeedScript applications are often stateless. The distinction between state-aware and stateless applications is discussed in 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. For more information, see Handling DISPLAY Output.
- 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 ABL 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`
- `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 Handling DISPLAY Output.

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.

Related Links

- Statements
- Comments
- Blocks
- Running procedures and Web objects
- Blocks and context
- Block context and resource scope
- Compile-time versus run-time code

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, as shown:

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 Customer.Balance GT 20000 THEN DO: /* BEGIN Non-iterative Block */
      {&DISPLAY} Customer.Name Customer.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.

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 *OpenEdge Development: ABL Reference*.

Related Links

- Persistent procedures
- WebSpeed calling conventions

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.

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 *OpenEdge Getting Started: ABL Essentials* or the *OpenEdge Development: ABL 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 `fBalance`, defined within a `FOR` block:

sample3

```
FOR EACH Customer FIELDS (Balance) NO-LOCK:
    DEFINE VARIABLE fBalance AS DECIMAL NO-UNDO.
    fBalance = fBalance + Customer.Balance.
END.
{&OUT} "<P>" "Corporate Receivables Owed:"
        STRING(fBalance, "$>,>>>,>>>,>>9.99") "</P>" .
```

However, `fBalance` 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.

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 Compile-time versus run-time code.) 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 `fBalance`, it can no longer access the `Customer.Balance` field for any record read into the `Customer` buffer by the `FOR` block.

Related Links

- Unscoped resources
- Dynamic resources

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 Customer FIELDS (Balance Name) NO-LOCK:
  DEFINE VARIABLE fBalance AS DECIMAL NO-UNDO.
  fBalance = fBalance + Customer.Balance.
  {&DISPLAY} Customer.Name Customer.Balance.
END.
FIND FIRST Customer NO-LOCK NO-ERROR.
IF AVAILABLE Customer THEN
  {&DISPLAY} Customer.Name 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 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).

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.

Related Links

- Compile-time code
- Compile-time syntax elements
- Run-time code
- Run-time syntax elements
- How compile-time and run-time code interact

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 components enforce the iteration conditions (when the block starts and stops executing) and select what database data is 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 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 FIELDS(Balance Name Phone) NO-LOCK
  WITH FRAME alpha SIDE-LABELS:
  {&DISPLAY} Customer.Name Customer.Phone Customer.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 (cProc[iProc]) to provide a compile-time object name:

sample6

```
DEFINE VARIABLE cProc AS CHARACTER NO-UNDO EXTENT 3
  INITIAL ["proc1.p", "proc2.p", "proc3.p"].
DEFINE VARIABLE iProc AS INTEGER   NO-UNDO.

{&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 iProc = 1 TO 3:
  RUN VALUE(cProc[iProc]).
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.

WebSpeed preprocessors

The preprocessor is a function of the Progress ABL 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 the following table, 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 1: WebSpeed preprocessors

Preprocessor name	Assigned value
`&WEBSTREAM`	`STREAM Webstream`
`&OUT`	`PUT {&WEBSTREAM} UNFORMATTED`
`&OUT-FMT`	`PUT {&WEBSTREAM}`
`&OUT-LONG`	`EXPORT {&WEBSTREAM}`
`&DISPLAY`	`DISPLAY {&WEBSTREAM}`

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.

Related Links

- Message handling
- General information exchange
- Passing information between Web requests
- Managing date and time information
- Checking configuration options
- Generating Web page headers

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. The following table lists these functions.

Table 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.

Related Links

- Generating messages directly
- Generating messages with a custom tag

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 to appear prior to the messages output for the specified group

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

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, as shown in this code:

```
<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">
```

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.

General information exchange

WebSpeed supports several generally useful API functions for information exchange, some of which are most commonly used to handle Web requests. The following table lists these functions.

Table 3: 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.

Function	Return type	Description
		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.

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.

Related Links

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

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 agent control program (web-disp.p) and the broker.

The following table lists the API functions that support cookie passing.

Table 4: Cookie-passing API functions

Function	Return type	Description
`delete-cookie`	`CHARACTER`	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. WebSpeed provides several API functions to facilitate the building and reading of URLs, listed in the following table. (The definitions for these functions reside in `install-path`/src/web/method/cgiutils.i.)

Table 5: 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) 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.

Function	Return type	Description
`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.

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 General information exchange.

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.

The following table lists the API functions that support hidden field construction. These functions construct complete HTML definitions for hidden fields in a form.

Table 6: 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 General information exchange.

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 the following table.

Table 7: 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`.

Managing date and time information

The functions listed in the following table convert dates and times between local and UTC time, and format dates and times for different types of Web output. (The definitions for these functions reside in `install-path`/src/web/method/cgiutils.i.)

Table 8: Date and time API functions

Function	Return type	Description
`convert-datetime`	`CHARACTER`	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.

Checking configuration options

The following table 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 9: 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.

Generating Web page headers

The functions listed in the following tables 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 10: 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`	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.

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 *OpenEdge Development: Programming Interfaces*, which describes XML support in the context of Progress ABL. However, the information also applies to SpeedScript, which is based on Progress ABL.

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 Customer.Name BEGINS "s" NO-LOCK:
  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 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 Using <META> and <!-- WSMETA --> tags.

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 How to embed SpeedScript in HTML.

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.

Note: Progress Developer Studio for OpenEdge is an Eclipse-based OpenEdge product that supports creating and managing WebSpeed applications. It contains full-featured editors for editing HTML, Embedded SpeedScript, and CGI Wrapper files. In many respects, OpenEdge Developer Studio is an excellent alternative to the AppBuilder and other tools for WebSpeed application development. For more information, see the Progress Developer Studio for OpenEdge help in the Development Tools section of the OpenEdge Product Documentation.

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.

Related Links

- Statement escapes
- Expression escapes

Statement escapes

The following table shows the supported statement escapes.

Table 11: Statement escapes

Start tag	End tag	Comments
<SCRIPT>	</SCRIPT>	This escape is supported by most authoring tools. Use the LANGUAGE attribute to specify a scripting language. For example, LANGUAGE="SpeedScript".
<?WS>	</?WS>	These are WebSpeed-defined tags. The <? sequence is an SGML directive, but some authoring tools might not support it.
<%	%>	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 Customer.Name BEGINS "s" NO-LOCK:
  DISPLAY {&WEBSTREAM} Customer.
END.
</?WS>
<HR>
</BODY>
</HTML>
```

Expression escapes

The following table shows the supported expression escapes.

Table 12: Embedded SpeedScript expression escapes

Start tag	End tag	Comments
`	`	The back tic (`) is acceptable to most 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 Customer.Name BEGINS "s" NO-LOCK: -->
<TR>
  <TD> `Customer.Cust-num` </TD>
  <TD> `Customer.Name` </TD>
  <TD> `Customer.Phone` </TD>
</TR>
<!--WSS END. -->
</TABLE></CENTER>
```

```
</BODY>
</HTML>
```

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 Embedded SpeedScript examples.

Related Links

- Passing parameters
- Managing dynamic pages with forms

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 Customer.Name BEGINS p_filter NO-LOCK: -->
<TR>
  <TD> `Customer.Cust-num` </TD>
  <TD> `Customer.Name` </TD>
  <TD> `Customer.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").
```

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.

Related Links

- Defining local data and getting input
- Defining forms
- Managing SpeedScript blocks and conditional execution
- Building URL query strings and closing out the page

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):

Part 1 of browse.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>" Customer.Name "</TD><TD>"~
    Customer.Cust-num "</TD></TR>"
  &SCOPED-DEFINE Result-Rows 10

  DEFINE VARIABLE jumpto-value        AS CHARACTER NO-UNDO.
  DEFINE VARIABLE {&Filter-Field}-key AS CHARACTER NO-UNDO.
  DEFINE VARIABLE repo-to             AS INTEGER   NO-UNDO.
  DEFINE VARIABLE iCount              AS INTEGER   NO-UNDO.
  DEFINE VARIABLE JumpForm            AS CHARACTER NO-UNDO.
  DEFINE VARIABLE TmpUrl              AS CHARACTER NO-UNDO.
```

```
  DEFINE VARIABLE DelimiterField      AS CHARACTER 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:

Part 2 of 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:

Part 3 of 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:

Part 4 of 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. */
    repo-to = IF NOT AVAILABLE {&Query-Table} THEN -1
              ELSE CURRENT-RESULT-ROW("Browse-Qry":U) + 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',?).
    DelimiterField = IF INDEX(TmpUrl,"?") GT 0 THEN ? ELSE "?".
</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:

Part 5 of 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>
```

Note that onMouseOver and onMouseOut are 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.

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.

Related Links

- Specifying file type options
- Specifying HTTP header information
- Generating information prior to HTTP header output

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 Specifying HTTP header information.

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.

In the following code, the file management procedure calls the preprocessor for meta.html, passing "get-options":

```
DEFINE VARIABLE speedfile AS CHARACTER NO-UNDO.
DEFINE VARIABLE wsoptions AS CHARACTER NO-UNDO.
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.

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 Specifying file type options.

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.

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.

Compiling and running embedded SpeedScript Web objects

Whatever authoring tool you use to create an embedded SpeedScript file, you must compile the file to generate a Web object. The AppBuilder allows you to create an embedded SpeedScript file and generate a Web object from it. Or, you can author the embedded SpeedScript file using another tool and compile it with either the AppBuilder or WebSpeed WebTools.

Related Links

- Compiling embedded SpeedScript Web objects
- Running embedded SpeedScript Web objects

Compiling 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 contains embedded SpeedScript, 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.

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 agent tries to locate the corresponding r-code file to execute and returns an error if it cannot find it.

4

Handling DISPLAY Output

This chapter describes how to manage the DISPLAY output from WebSpeed Web objects.

Related Links

- Working with DISPLAY output
- Changing output format defaults

Working with 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.

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

Related Links

- Directing DISPLAY output
- Formatting DISPLAY output

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 manage 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.

Related Links

- Using the WebSpeed-defined output stream
- Defining and using a new output stream

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 NO-LOCK.DISPLAY {&WEBSTREAM} Customer.Name Customer.Phone.
{&DISPLAY} Customer.Name Customer.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 NO-LOCK.
DEFINE STREAM MyStream.
OUTPUT STREAM MyStream TO "WEB".
DISPLAY STREAM MyStream Customer.Name Customer.Phone.
```

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

```
FIND FIRST Customer NO-LOCK.
OUTPUT TO "WEB".
DISPLAY Customer.Name Customer.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 NO-LOCK.
OUTPUT TO "WEB".
DISPLAY Customer.Name Customer.Phone.
DISPLAY {&WEBSTREAM} Customer.Address Customer.City Customer.PostalCode.
```

For more information on the statements used in these examples, see *OpenEdge Development: ABL Reference*.

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 *OpenEdge Development: ABL 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

Related Links

- Paging options of the OUTPUT TO "WEB" statement
- HTML attributes of the WEB-CONTEXT handle

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.

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. This divider is specified by the HTML-END-OF-PAGE attribute of the WEB-CONTEXT system handle. For more information, see HTML attributes of the WEB-CONTEXT handle.

For more information on setting these options, see Changing output format defaults.

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.

The following table describes how to use these attributes. For information on how to set and read these attributes, see the WEB-CONTEXT Handle entry in *OpenEdge Development: ABL Reference*. For information on setting new defaults for these attributes, see Setting HTML attribute defaults.

Table 13: WEB-CONTEXT handle HTML attributes

Attribute	Output location	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 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.
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.

Attribute	Output location	Description
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.
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.
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.

Changing output format defaults

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

Related Links

- Setting {&WEBSTREAM} defaults
- Setting HTML attribute defaults

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 Formatting DISPLAY output.) 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.

Setting HTML attribute defaults

You can change the initialized defaults for all the HTML attributes of the WEB-CONTEXT handle. (See Formatting DISPLAY output.) 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.

Related Links

- Entering attribute definitions
- Examples

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 the following elements in order, and with no intervening spaces (except as noted).

Element	Description
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 new lines. 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

Element	Description
	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.

5

Generating HTML Visualizations

This chapter describes an implementation of HTML table generation installed with WebSpeed as a prototype for HTML visualizations.

Related Links

- Overview
- Generating HTML tables with a custom tag
- Generating HTML tables directly from SpeedScript
- Adding a template to AppBuilder
- Support for other HTML visualizations

Overview

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 SpeedScript.

Related Links

- Using the support for HTML table visualizations

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 WebSpeed 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.

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.

Related Links

- Rules for using custom tags
- Using the <!--WSTAG--> custom tag
- How tagrun.p interacts with the custom tag
- Using the table template
- Compiling the table template

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.

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 Progress® WebSpeed® 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 the NAME and FILE attributes in order to generate a table or other custom HTML output.

Related Links

- NAME attribute
- FILE attribute

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.

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.

The web.output trigger 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.

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.

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.

Compiling the table template

Because the table template is an embedded SpeedScript file, you must compile it to generate a r-code 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-code (.r) file in the custom tag in the HTML file:

```
<!--WSTAG NAME="mytbl" FILE="mytbl.r" -->
```

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.

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*.

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.

6

Using JavaScript with WebSpeed

This chapter contains information for developers who want incorporate JavaScript into their WebSpeed applications.

Related Links

- SpeedScript versus JavaScript
- Using JavaScript source files
- Some JavaScript examples

SpeedScript versus JavaScript

It is a common practice to use both SpeedScript and JavaScript when developing WebSpeed applications. SpeedScript has advantages for developing the business logic of a applications, while JavaScript is a good programming tool for adding user interface elements to Web applications.

If you use either the Report or Detail Wizards in AppBuilder to create a WebSpeed Web object, you can view the resulting HTML source file and see a combination of SpeedScript and JavaScript. The wizards create SpeedScript to implement database queries and updates, and they create JavaScript event handlers (like `onMouseOver`, `onClick`, etc.,) to implement interactive features of WebSpeed applications.

The <SCRIPT> tag for JavaScript employs the same syntax as the <SCRIPT> tag for Embedded SpeedScript, as shown:

```
<SCRIPT LANGUAGE="JavaScript">
JavaScript Code
</SCRIPT>
```

In some situations, you do not need a <SCRIPT> tag. JavaScript event handlers, for example, do not require a <SCRIPT> tag when they are used as an attribute to an HTML tag, as shown:

```
<BODY onLoad="alert('Done');">
```

Some other factors that you should keep in mind when using JavaScript in WebSpeed applications are:

- End users of your WebSpeed application will be able to see your JavaScript code when they view HTML source in their browsers. They can see the HTML output that Embedded SpeedScript generates, but they do not see the actual SpeedScript source code. (This is because the SpeedScript code executes on the server side while the JavaScript executes on the client-side browser.)
- No static or dynamic HTML can be generated from the JavaScript code that is between HTML <SCRIPT> tags.
- SpeedScript is executed on the server side by the WebSpeed agent. JavaScript is executed on the client side by the Web browser.

Using JavaScript source files

If you develop JavaScript routines that are long and that are common to a number of applications, you may want to keep them in separate JavaScript source (.js) files.

When deploying JavaScript source files, you must put them in the Web server's root directory or in a subdirectory of the Web server's root directory. You can then reference the file using the HTML SRC attribute. For example, the following calls myscript.js, which is in jsscripts a subdirectory of the Web server's root directory:

```
<SCRIPT LANGUAGE="JavaScript" SRC="/jsscripts/myscript.js">
```

You can also create references to JavaScript source files within Embedded SpeedScript. However, you must be careful to specify the complete <SCRIPT> tag expression and to escape the expression using back tics. For example, the following code snippet **will not** run the JavaScript file:

```
<SCRIPT LANGUAGE="SpeedScript">
  DEFINE VARIABLE myVar AS CHARACTER NO-UNDO.
  myVar = "/jsscripts/myscript.js".
</SCRIPT>

<H3>Trying to run JavaScript ...</H3>
<SCRIPT LANGUAGE="JavaScript" SRC="`myVar`">
```

```
</SCRIPT>

<H3>JavaScript did not run...</H3>
```

The problem is fixed when you assign the complete <SCRIPT> tag expression to the variable and enclose it in back tics, as shown:

```
<SCRIPT LANGUAGE="SpeedScript">
  DEFINE VARIABLE myVar AS CHARACTER NO-UNDO.
  myVar = '<SCRIPT LANGUAGE="JavaScript"
    SRC="/jsscripts/myscript.js"></SCRIPT>'.</SCRIPT>
<H3>Now running JavaScript ...</H3>
`myvar`
<H3>JavaScript done ...</H3>
```

Some JavaScript examples

It is beyond the scope of this book to give you detailed instructions on using JavaScript. However, this section gives a few examples that you might find useful in your WebSpeed applications.

Related Links

- A combo box with URL buttons
- Browser detection
- Printing from an HTML page

A combo box with URL buttons

This example shows you how to code a combo box to hold URLs. When the user clicks on the button, the browser goes to the Web site specified in the URL:

```
<FORM>
  <SELECT SIZE="1"
    onChange="(this[selectedIndex].value ?
              location=this[selectedIndex].value : null)">
    <OPTION SELECTED>Select Location
    <OPTION VALUE="http://www.progress.com">Progress
    <OPTION VALUE="http://www.progress.com/services/support">Tech Support
    <OPTION VALUE="http://www.webspeed.com">WebSpeed
  </SELECT>
</FORM>
```

Browser detection

The following JavaScript code snippet contains a simple method for determining whether a browser is Internet Explorer or Netscape Navigator. It tests for the existence of the document.all object, which is only

available on Internet Explorer. Then it tests for the existence of the document.layers object, which is only available on Netscape Navigator, as shown:

```
var isIE = false;
var isNav = false;
if (document.all)
  isIE = TRUE;
else if (document.layers)
  isNav = TRUE;
```

An alternative method for browser detection is to use SpeedScript to create a test in your WebSpeed web object which, of course, runs on the server side. The following test, written in SpeedScript, checks for the value of the CGI environment variable, `HTTP_USER_AGENT`:

```
IF INDEX(get-cgi('HTTP_USER_AGENT':U)," MSIE ":U) GT 0 THEN isIE = TRUE.
```

CGI environment variables are set automatically when web requests are received by a WebSpeed agent. In this case, if the value of `HTTP_USER_AGENT` is `MSIE`, then the client request came from an Internet Explorer Web browser.

Printing from an HTML page

This example shows how a Web page can print itself after it finishes loading in the Web browser:

```
<BODY onLoad="window.print()">
This example will pop up the printer dialog box when it's loaded into the
browser. The intention of this example is for WebSpeed users to create
applications that can print HTML documents without browser interaction.
</BODY>
```

The next example shows how you can print a portion of a page dynamically. First, you must set up two frames, one that takes up the entire browser window and another that is hidden, as shown:

```
/* startup.w */
<FRAMESET FRAMEBORDER=1 COLS="100%,*">
  <FRAME ID="appFrame" SRC="myApp.w" SCROLLING="no">
  <FRAME ID="jsFrame" SRC="empty.html" SCROLLING="no"
    NORESIZE FRAMEBORDER=0 MARGINHEIGHT=0 MARGINWIDTH=0>
</FRAMESET>
```

The `appFrame` contains the application. The `jsFrame` is used for communicating with the server and for running applications like printing.

The following shows the `filePrint()` function in the simplified `myApp.w code` fragment:

```
/* myApp.w fragment */
<SCRIPT LANGUAGE="JavaScript">
```

```
function filePrint() {
  var newPage = '<PRE>' + document.form1.elements["textarea0"].value;
  parent.jsFrame.document.write(newPage);
  parent.jsFrame.document.close();
  parent.jsFrame.focus();
  parent.jsFrame.print();
}
</SCRIPT>
```

The `filePrint()` function copies the value of a `<TEXTAREA>` field named `textarea0` to the hidden frame with a `<PRE>` prefix. Focus is moved to that frame and then it is printed. If you eliminate the `<PRE>` prefix, the browser renders the context of `textarea0` before it is printed.

7

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,

Related Links

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.

Related Links

- Web object states
- WebSpeed application states
- Advantages and drawbacks of state-persistence

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.

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 state-persistent transactions.

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

Understanding state-persistent transaction control

State-persistent transactions allow you to maintain active context on a single WebSpeed agent between requests by locking that agent to a single client. To support state-persistent 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 transaction, locks the agent on behalf of the Web object that starts a transaction, and unlocks the agent on behalf of any Web object that terminates the transaction.

To lock an agent temporarily to a particular browser, 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 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 state-persistent transaction, requests made to the same URL by other client 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.

Related Links

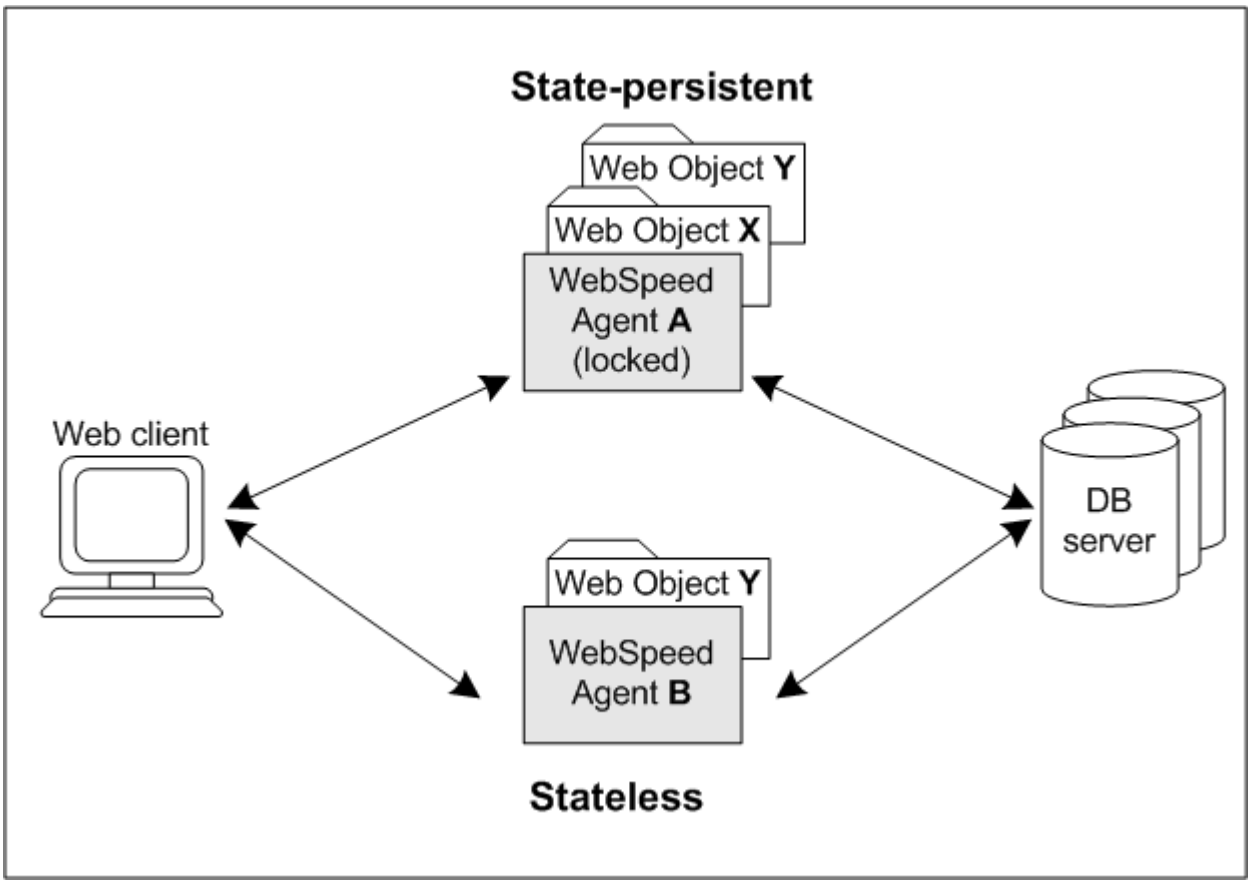
How to make a Web object state-aware

You run the `setWebState` function to make a Web object state-aware, or to cancel the state-aware condition. The setWebstate procedure must be run before the HTTP header output and should be run before any cookie functions are called. See setWebState. For more detailed information about controlling WebSpeed transactions, see Implementing transaction control.

Web objects in stateless and state-persistent contexts

The following figure 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 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 the figure, 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 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.

Primary and secondary Web objects

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.

Related Links

- Running primary and secondary Web objects
- State-aware Web objects and persistence

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 client must make the next transaction request back to the primary Web object.

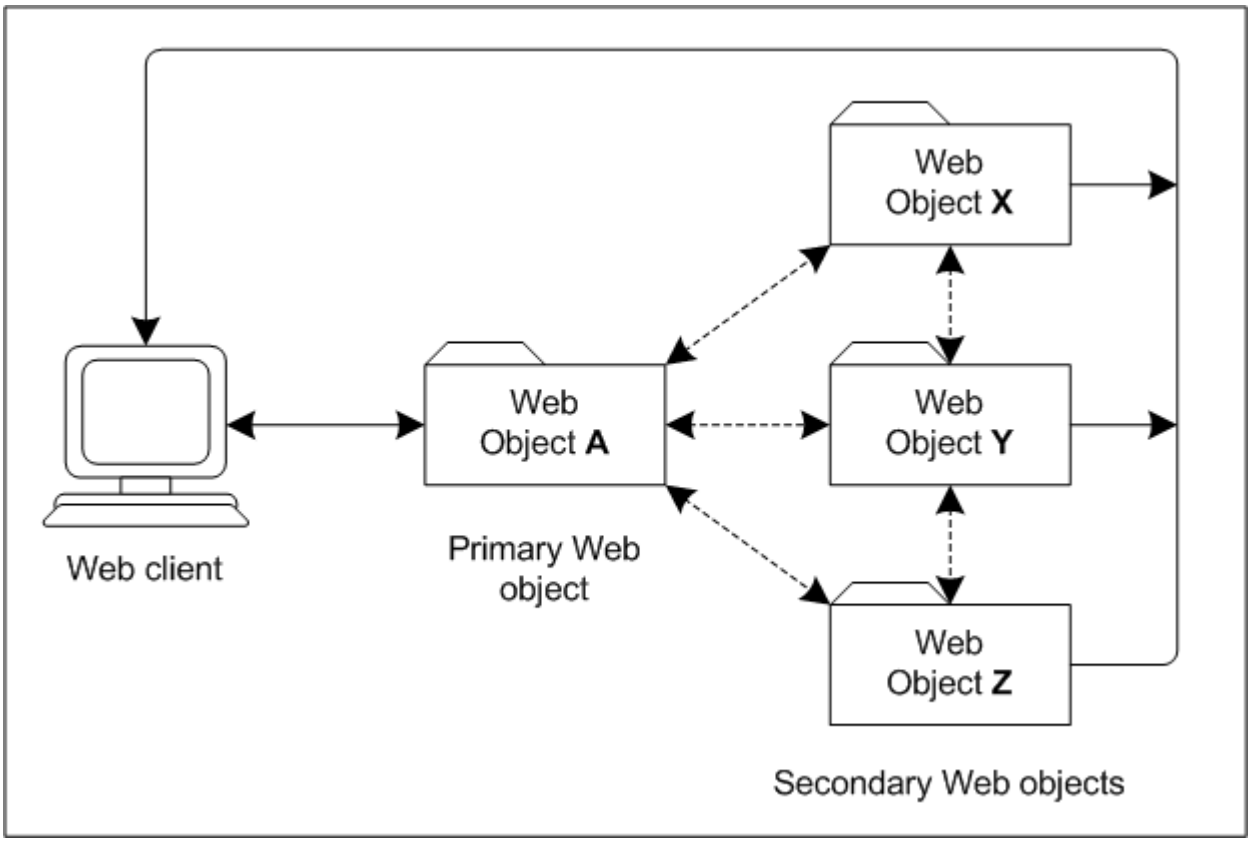
The primary Web object typically sets the time-out period for the WebSpeed transaction. As the primary and secondary Web objects respond to requests from the client, the time-out counts down. The time-out does not reset unless the primary Web object explicitly resets it by calling setWebstate. Note that 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.

The following figure shows the relationship among a 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 1. Primary and secondary Web objects in a WebSpeed transaction



The request cycle shown in the figure illustrates these points:

1. The 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.

State-aware Web objects and persistence

You might have noticed that the scenario in Running primary and secondary Web objects 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.

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 Transaction control with CGI Wrapper Web objects and Transaction control with HTML-mapping Web objects.

Related Links

- Modifying the outputHeader procedure
- Modifying the process-web-request procedure
- Overriding procedures and functions
- Modifying the attribute list
- Modifying the user field list

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).
```

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 Resetting a Web object's time-out period.

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 Passing information between Web requests. For syntax information, see set-cookie.

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 Anatomy of process-web-request in HTML-mapping.

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 Handling state-aware time-outs.)

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 *OpenEdge Development: ABL 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 *OpenEdge Development: ABL Reference*. Also see the AppBuilder online help.

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 Handling state-aware time-outs.

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. For an example of a user field used to manage a WebSpeed transaction, see Handling state-aware time-outs. 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.

Transaction control and embedded SpeedScript 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 SpeedScript.

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 Implementing transaction control), 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 part 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.

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 Implementing transaction control), 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 part 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.

Related Links

- Anatomy of process-web-request in HTML-mapping
- Modifying the request logic
- Moving data through the HTML mapping Web object
- Creating Definitions
- Modifying the Main Code Block
- Customizing field object control handlers

Anatomy of process-web-request in HTML-mapping

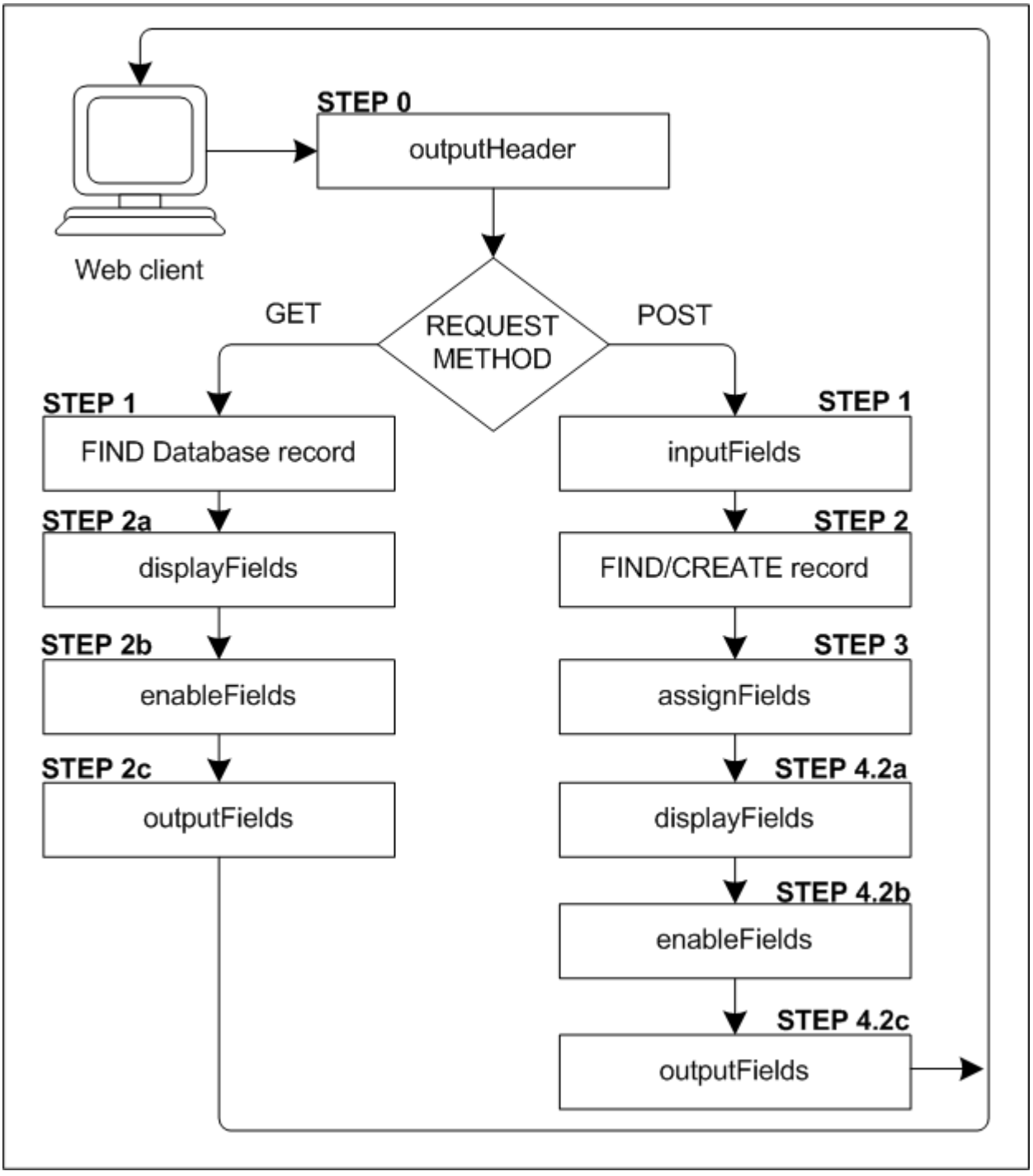
The heart of an HTML-mapping Web object is `process-web-request`. The following figure shows how the default HTML-mapping version of `process-web-request` participates in a Web request.

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 box 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` — An 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 1. Standard HTML-mapping process-web-request model



Note: Step 4.1 is missing in the figure above. 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.

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 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.

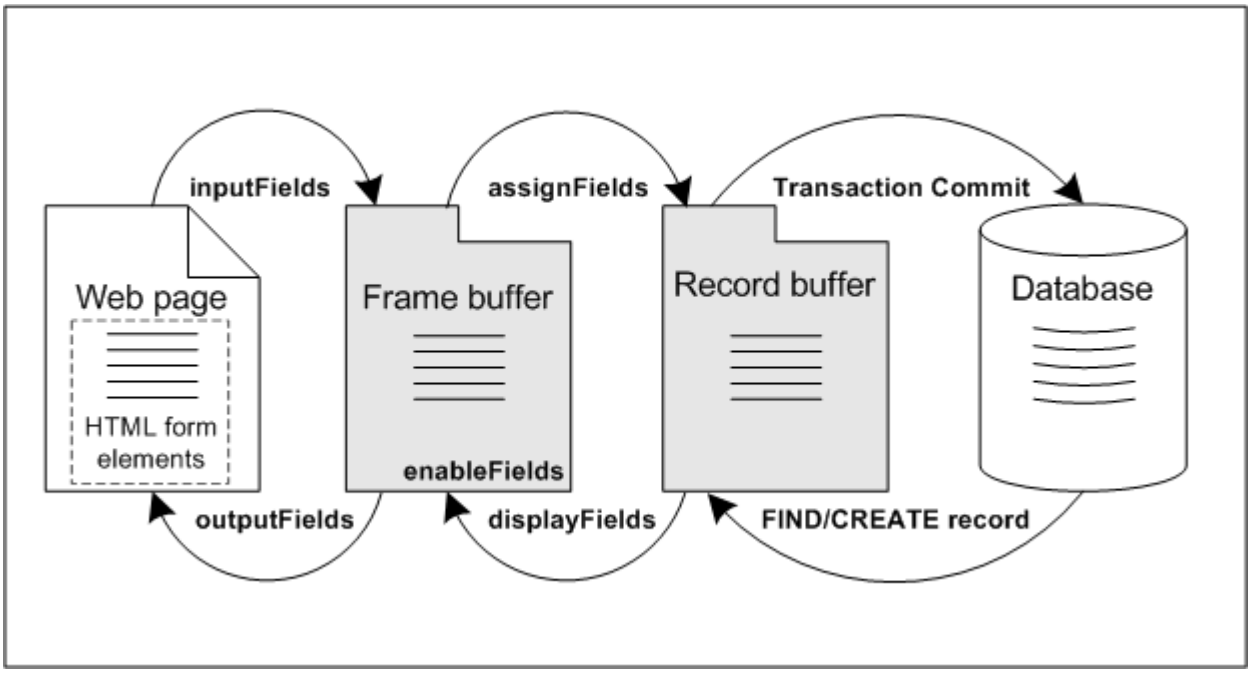
Because Web objects do not care where their input comes from, you can write a WebSpeed application that bases its request logic solely on the values returned by the get-value() API function and ignore the inputFields method entirely or provide a local inputFields override.

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.

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, the following figure shows where the default event procedures move data in a Web object.

Figure 1. Data flow through an HTML-mapping Web object



Related Links

- Frame buffer and record buffer
- inputFields data movement
- assignFields data movement
- Committed transactions
- FIND records
- displayFields data movement
- enableFields effects
- outputFields data movement

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 ABL. The difference is that for ABL, 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 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.

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 Running procedures and Web objects.

You can add any parameter, variable, or preprocessor definitions that you need for your Web object in this section.

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, as shown:

```
/* 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 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 WebSpeed 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.

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
#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.

`typ-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 *OpenEdge Development: ABL 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 tags in an HTML file, as shown:

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`.

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.

Related Links

- Using a Web object time-out handler
- Resetting a Web object's time-out period

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`.

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, as shown:

```
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.

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.

Advantages of using SERVER-CONNECTION-ID

Using the SERVER-CONNECTION-ID attribute allows you to create a virtual session for your WebSpeed applications. You can maintain context between Web requests without tying up a WebSpeed Agent.

As described in Defining state, Web requests can be sent in a stateless mode or in a state-aware mode. In the stateless mode, web requests are independent of each other. No application context is maintained from one request to another. In state-aware mode, context is maintained by locking a WebSpeed agent to a particular client. State-aware mode has a scalability disadvantage because the locked agent is not available until a session is completed.

When you enable Session Connection ID, the WebSpeed Messenger checks incoming requests for a cookie containing a value for SERVER-CONNECTION-ID. The SERVER-CONNECTION-ID variable contains a unique identifier. If it does not exist, the WebSpeed Messenger causes a unique identifier to be generated. The WebSpeed Messenger then sets the SERVER-CONNECTION-ID attribute of the SESSION handle to the value of the unique identifier. (For more information, see *OpenEdge Development: ABL Reference.*) This unique connection identifier gets passed to the WebSpeed Agent as part of the Web request. In turn, the WebSpeed Agent passes the unique connection identifier to the WebSpeed application (also known as the Web object).

It is important to be aware that your Web object **will not** automatically associate the current request with a prior request even though it receives a unique connection identifier. You will need to create a data source where you can save connection identifiers and their associated application context. Then, you must create the logic in the Web object that:

- Checks for SESSION: SERVER-CONNECTION-ID
- Determines if the connection identifier is new or if it already exists in your data source
- Creates a new entry in your data source if the connection identifier is new
- Restores application context if the connection identifier already exists in your data source

- Saves the changes in application context with the appropriate connection identifier before it completes execution

Enabling connection identifiers

The WebSpeed Messenger must be enabled to check for connection identifiers and to create them if they do not already exist. The easiest way to do this is to modify the WebSpeed Messenger properties in OpenEdge Management or OpenEdge Explorer.

To enable connection identifiers:

1. Expand the **Messengers** node under **OpenEdge** in OpenEdge Management or OpenEdge Explorer.
2. Select the WebSpeed Messenger that you want to configure (**CGIIP**, **WSASP**, **WSISA**, or **WSNSA**).
3. Click **Configuration** in the **Command and control** section of OpenEdge Management or OpenEdge Explorer.
4. Open the **Advanced** tab.
5. Click **Edit**.
6. Select the **Session connection ID** check box.
7. Click **Save**.

Restart the Web Server if the Messenger is WSASP, WSISA, or WSNSA. You do not have to restart the Web Server for CGIIP Messengers.

Handling Binary Large Objects (BLOBs)

A binary large object (BLOB) contains unstructured data and consists of a group of bytes. ABL has no special knowledge about its contents. Rather, ABL assumes that the data in a BLOB is managed by other software (such as a word processor, a spreadsheet program, or a bar-code reader).

In the ABL, a BLOB is a data type used to specify a database table or temp-table field that contains a BLOB locator, which points to the associated BLOB data stored in the database. You must use a MEMPTR to manipulate the binary contents of a BLOB field in ABL.

WebSpeed applications can upload BLOBs and include them in their output.

To protect your system from malicious code, you should not allow unrestricted processing of uploaded binary files. Therefore, WebSpeed does not automatically store any file uploaded as part of a Web request. Uploaded files are held in memory. Your application can then manipulate the file through a `MEMPTR`.

The following code is an example of how to create a form that can upload a BLOB specified as a file:

```
<HTML>
  <BODY>
   <FORM ENCTYPE="multipart/form-data"
     ACTION="http://yourhost/msngr_path/msngr/target_app" METHOD="POST">
      <INPUT type="file" name="attachment1"> <INPUT type="submit">
```

```
    </FORM>
  </BODY>
</HTML>
```

To enable a WebSpeed application to accept data from this form, you must do the following:

- Verify that the WebSpeed Messenger is from OpenEdge 10.1A or a later release. The Messengers in previous versions cannot accept BLOBs.
- Specify a maximum size for uploaded files.
- Write code to manipulate the uploaded file through its `MEMPTR`. This code should ensure that the uploaded file contains no malicious content before it is stored to disk. WebSpeed provides a new API function to return the file's `MEMPTR`. See get-binary-data for the API's description.

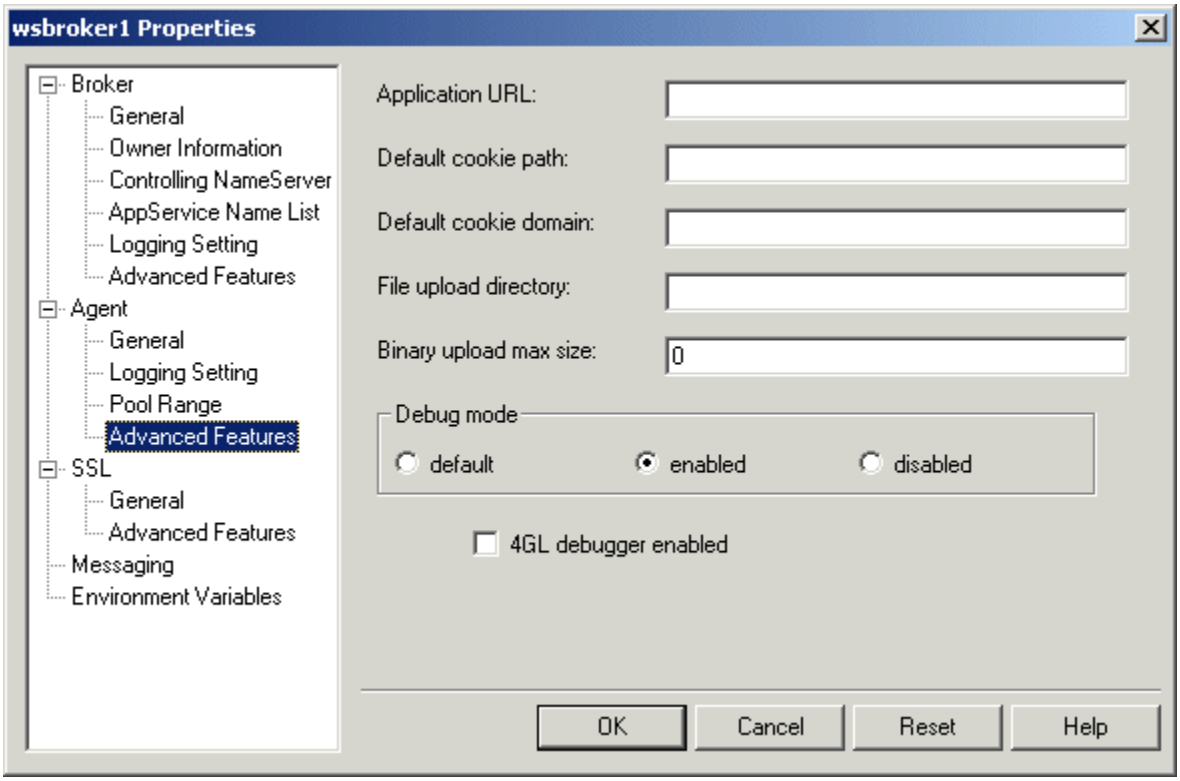
Related Links

- Specifying maximum size of uploaded files

Specifying maximum size of uploaded files

Uploading a file as part of a Web request ties up the WebSpeed agent that processes the request. This might cause performance problems for your application if several large files are uploaded at the same time. The `BLOB` data type has a maximum size of 1 GB, so a single upload could lock an agent for a significant period.

To control this problem, you can specify a maximum size for uploaded BLOBs. The maximum size is set with the `binaryUploadMaxSize` property in the ubroker.properties file. You can set it on the **Agent > Advanced Features** page of the **WebSpeed Transaction Server's Properties** sheet, as shown:



The following table lists the results of various settings for the upload maximum size.

Table 14: binaryUploadMaxSize settings

Value	Result
Any negative value	The agent does not limit the size of uploaded BLOBs.
0	The agent does not accept uploaded BLOBs.
1 byte – 1GB	The agent accepts uploaded BLOBs up to this size.

Note: The limit applies separately to each file received in a single Web request. If your Web page can accept more than one file in a particular Web request, you should consider the performance impact of several files of that size being uploaded together.

8

Controlling Database Transactions

This chapter describes how WebSpeed supports database transactions.

Related Links

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.

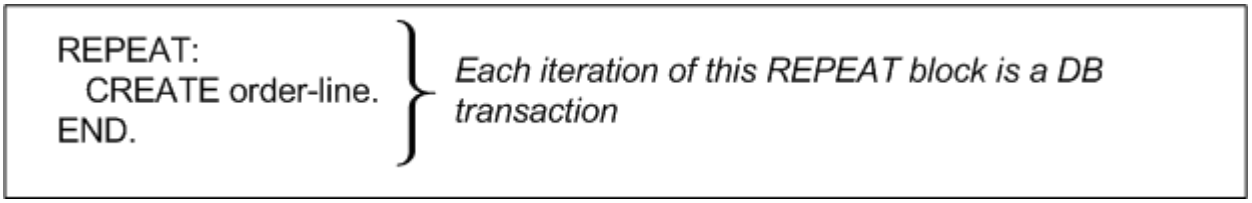
Related Links

- DB transactions in code
- All-or-nothing processing

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 the following, each order-line you create is a database transaction:

Figure 1. Database transaction definition

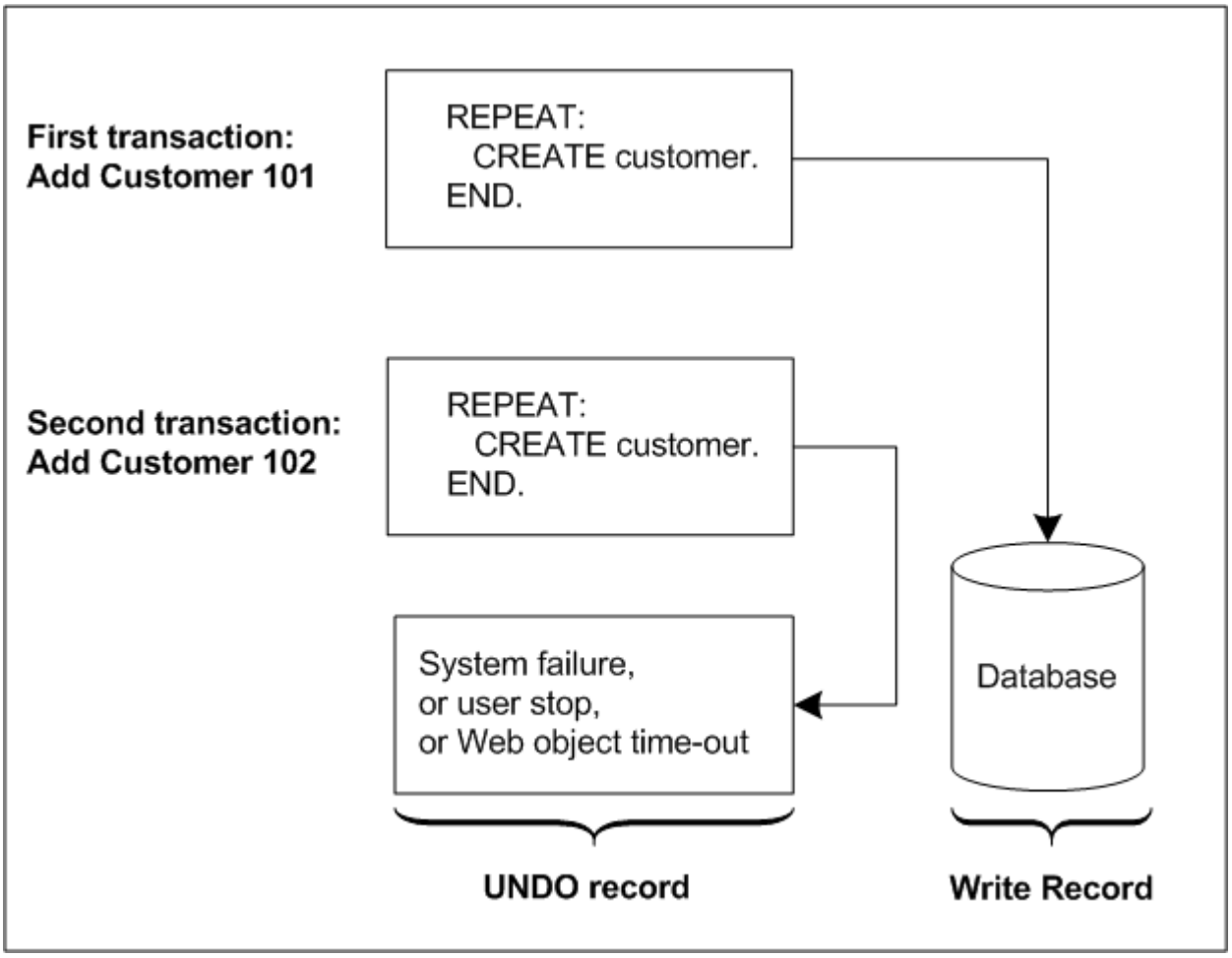


The database transaction is undone (or backed out) if:

- The system goes down (or crashes).
- The user presses the **STOP** button in a browser.
- A state-aware Web object times out.

In all of these cases, WebSpeed undoes all work it performed since the start of the database transaction, as illustrated in the following figure:

Figure 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.

All-or-nothing processing

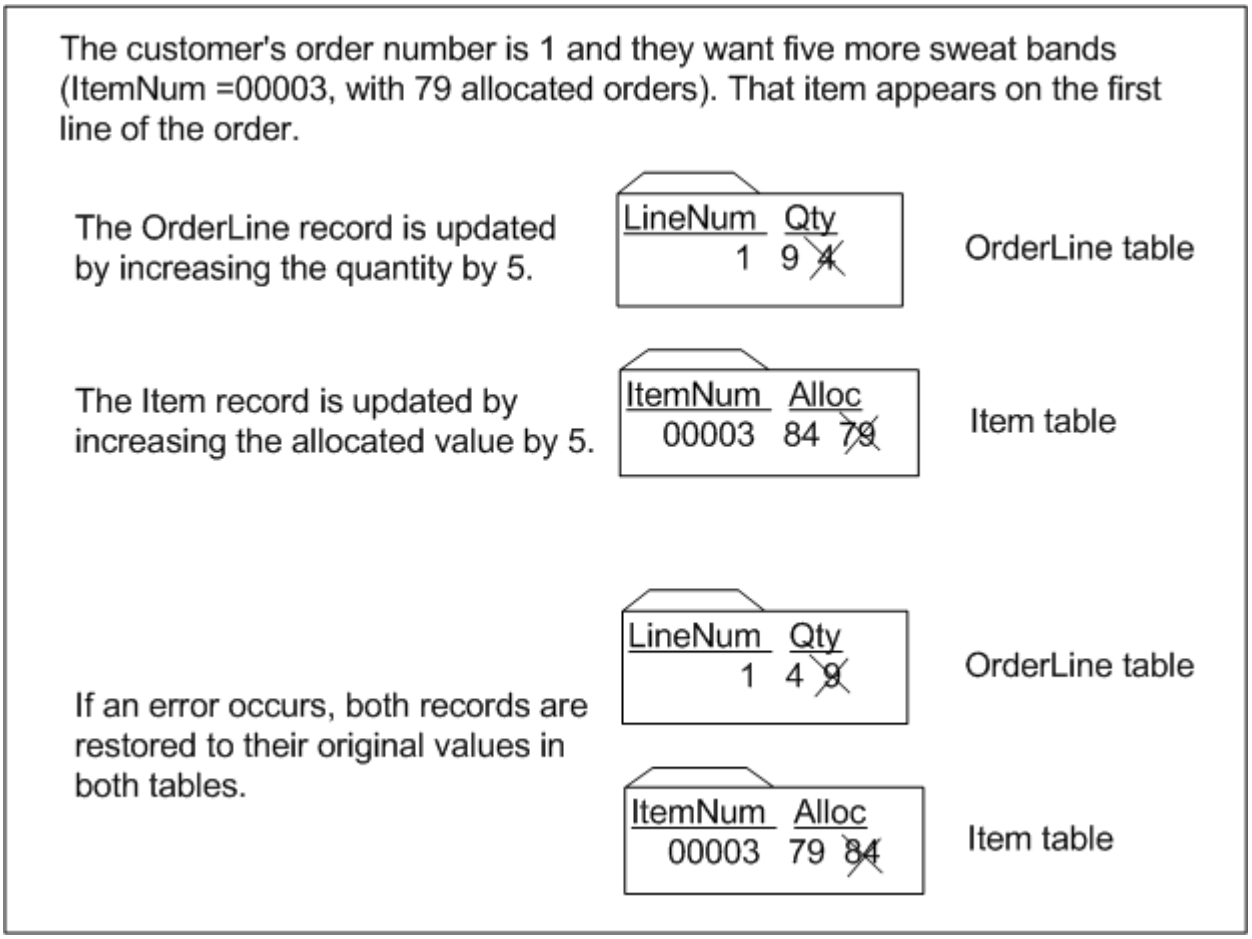
Suppose a customer calls to change an order for four sweat bands to nine sweat bands. This means you must make two changes to your database:

- 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.

The following figure shows this scenario for a database similar to Sports2000 (same tables, different data).

Figure 1. Database transaction involving two tables



Understanding the scope of database 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.

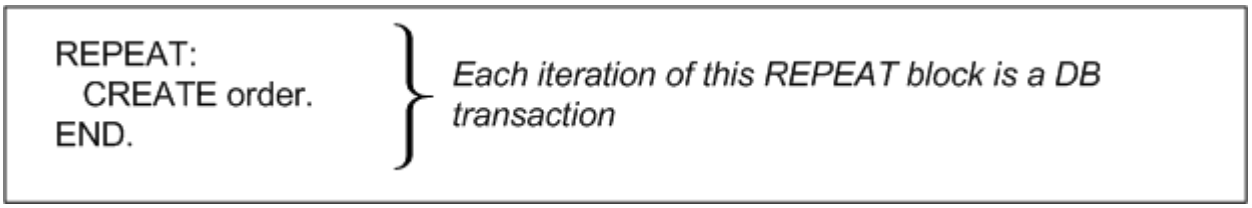
Directly updating the database means that the block contains at least one statement that can change the database. `CREATE`, `DELETE`, and `UPDATE` are examples of such statements.

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 following procedure has two blocks: the procedure block and the REPEAT block.

Figure 1. Database transaction scope



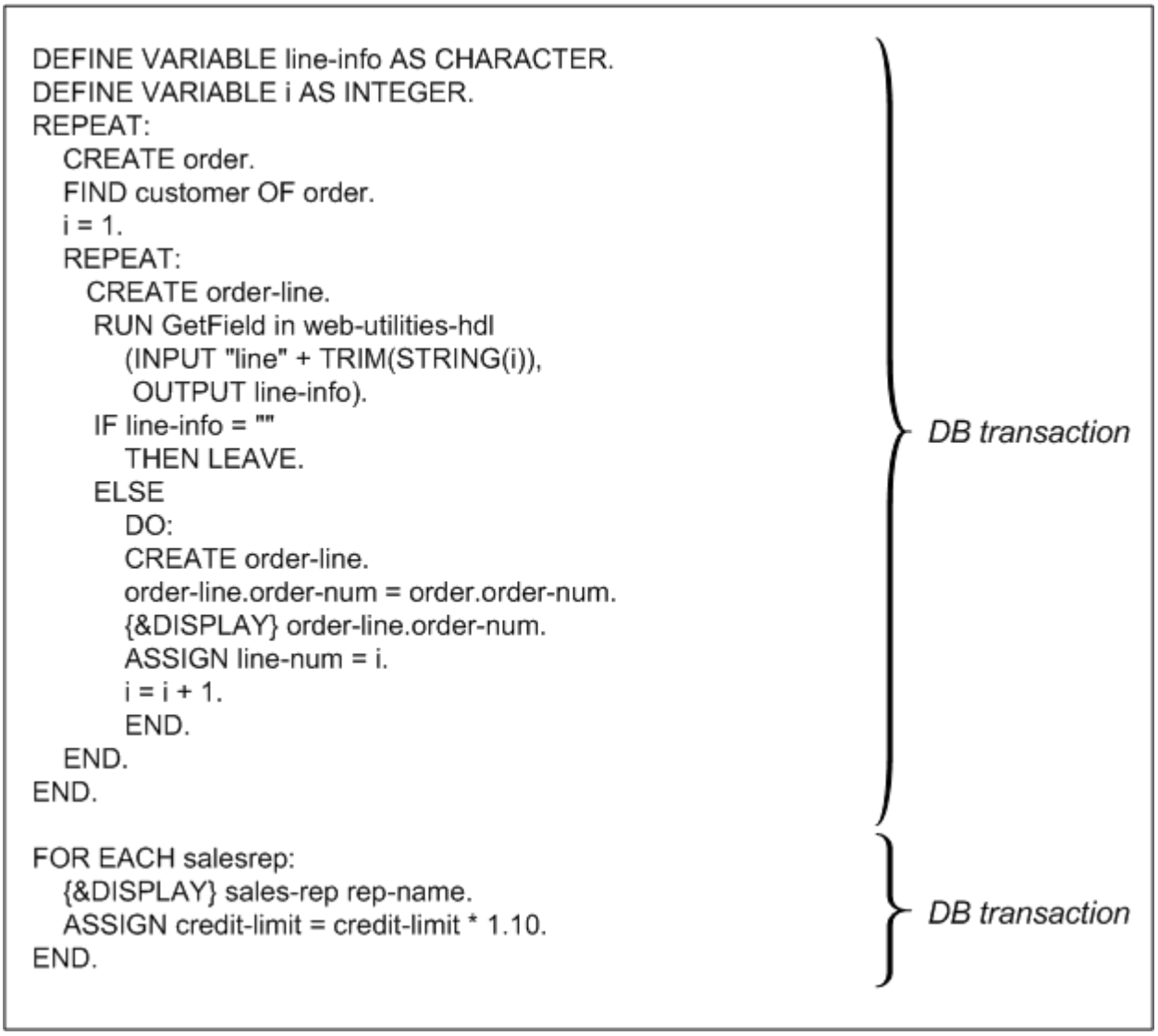
The procedure block has no statements directly in it that are not contained within the REPEAT block. The REPEAT block contains a CREATE statement that lets you add order records to the database. Because the REPEAT block is the outermost block that contains direct updates to the database, it is the transaction 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.

The following figure shows a procedure with multiple transactions.

Figure 2. Multiple DB transactions in a procedure



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.

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.

Related Links

- Subtransactions
- Controlling where DB transactions begin and end

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.)

The following table shows when database transactions and subtransactions are started.

Table 15: Starting database transactions and subtransactions

Type of block	Inactive transaction	Active transaction
`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	Does not start a subtransaction or a transaction	Starts a subtransaction

Type of block	Inactive transaction	Active transaction
either modify the database or read records using an EXCLUSIVE-LOCK.		

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.

Controlling where DB transactions begin and end

You might find that for certain procedure types, 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.

SpeedScript components and database transactions

Database transactions have varying effects on different SpeedScript components. This section describes the effects on:

- Subprocedures
- File input
- Program variables

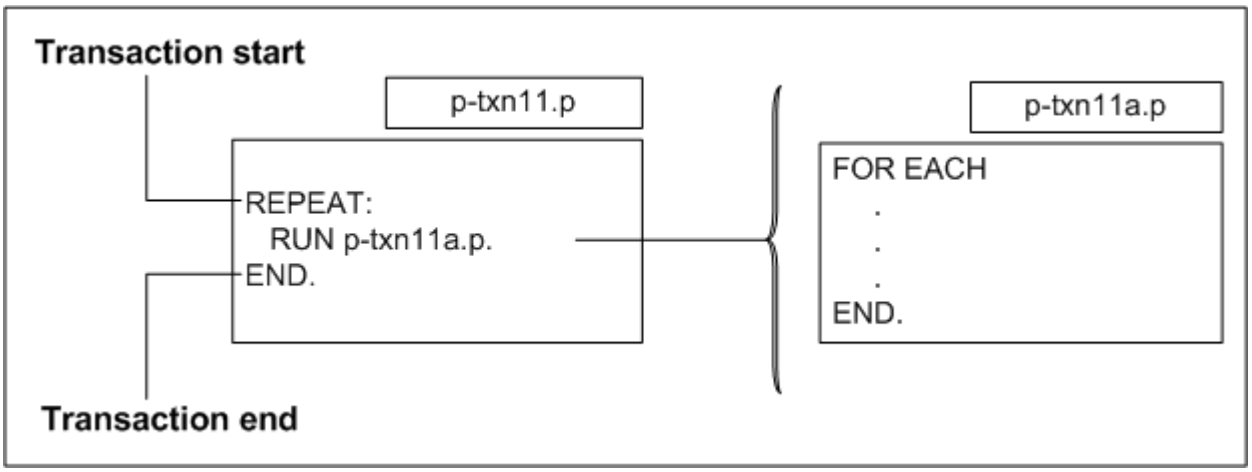
Related Links

- Subprocedures
- File input
- Program variables

Subprocedures

If you start a database transaction in a main procedure, that transaction remains active even while the main procedure runs called procedures. In the following figure, p-txn11.p runs p-txn11a.p within a database transaction.

Figure 1. 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.

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.

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.

Database 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 AppServers (distributed applications), each application component (WebSpeed agent and AppServer) controls separate database transactions.

Related Links

- Multi-database applications
- Distributed applications

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.

Related Links

- Two-phase commit
- Checking database connections

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 *OpenEdge Data Management: Database Administration*.

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 *OpenEdge Development: ABL Reference*.

Distributed applications

When a requesting application with an active transaction runs a remote procedure, the transaction is not propagated 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 *OpenEdge Application Server: Developing AppServer Applications*.

Determining when database 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 *OpenEdge Development: ABL Reference* for more information on the COMPILE statement and the TRANSACTION function.

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 *OpenEdge Data Management: Database Administration*.

Related Links

- Database transaction mechanics
- Subtransaction mechanics

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.

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 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.

Efficient database 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.

Multi-page database 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 WebSpeed 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.

Related Links

- Managing multi-page DB transactions
- Working with the multi-page transaction example
- Guidelines for usage

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 WebSpeed 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 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.

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**.

To start and terminate multi-page database transactions:

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.

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.
- For `RUN set-transaction-state ("START")` note:
  - 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 EXCLUSIVE-LOCK.
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 time-out. This will also force an "UNDO" of the multi-page transaction.

9

Debugging Applications

This section describes some debugging techniques that you can use to debug WebSpeed applications.

Related Links

- Overview
- Adding debug=on to URL
- Persistent debugging using cookies
- Calling the virtual debug Web object
- Debugging and administrative options
- Utilizing debugging in your application

Overview

When debugging is enabled, debugging information is appended to each Web page that your application generates. When 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 *OpenEdge Application Server: Administration*.

You can also use the Application Debugger to debug SpeedScript applications running on a WebSpeed agent. The process is the same as debugging an application running on an AppServer agent. For more information see *OpenEdge Development: Debugging and Troubleshooting* and the debugging chapter of *OpenEdge Application Server: Developing AppServer Applications*.

If you have Progress Developer Studio for OpenEdge installed, note that, in addition to the standalone OpenEdge Debugger that is described in this book, OpenEdge provides remote debugging support in the

Debug perspective of Progress Developer Studio for OpenEdge. The Debug perspective is an integral part of Progress Developer Studio for OpenEdge and works directly with the ABL Editor, making it easy to identify and fix problems as you work. For more information about using the Debug perspective to debug WebSpeed applications, see the Progress Developer Studio for OpenEdge online help.

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.

Persistent debugging using cookies

This debugging capability works best if you are using a browser that supports Persistent State Cookies. It is very easy to determine if your browser supports cookies or not. This is outlined in Debugging and administrative options.

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.

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.

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 Miscellaneous variables 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.

Related Links

- Agent specific information

- Persistent state cookies
- HTTP headers sent
- Miscellaneous variables
- Environment variables
- Form fields

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 Environment variables).

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.

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`, or `admin` (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.

Related Links

- Reading the broker's log files

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 *OpenEdge Data Management: Database Administration*.

10

WebSpeed API Reference

This section describes the PUBLIC functions and procedures that constitute the WebSpeed API. Any APIs in the WebSpeed source files that are not listed here should be considered PRIVATE. They might be radically restructured or removed at any time. There will be no formal notification of changes to PRIVATE APIs.

Note: All the paths in these entries are relative from `install-dir`/src.

Related Links

1. available-messages
2. check-agent-mode
3. convert-datetime
4. delete-cookie
5. format-datetime
6. getAttribute
7. get-binary-data
8. get-cgi
9. get-cgi-long
10. get-config
11. get-cookie
12. get-field
13. get-long-value
14. get-message-groups
15. get-messages

available-messages

This function returns `TRUE` if there are any messages queued for a specified message group, or for all message groups.

Location

web\method\message.i

Parameters

INPUT p_grp AS CHARACTER

A known message group into which messages may have been queued with the queue-message API function. The `Unknown value (?)` matches all message groups.

Returns

LOGICAL

Notes

All queued messages are output after the Web object specified in the URL returns. As such, messages do not remain queued between requests.

Examples

See the `PrintVars` procedure in web\support\printval.p and the main block in web\objects\web-disp.p.

See also

get-messages

get-message-groups

output-messages

queue-message

Next topic: check-agent-mode

check-agent-mode

This function returns `TRUE` if the agent is running in the specified mode (development or production).

Location

web\objects\web-util.p

Parameters

INPUT p_mode AS CHARACTER

The environment mode of the transaction server, either "production", "development", or "evaluation".

Returns

LOGICAL

Notes

This function does not compare for equality because the environment mode setting might contain other, user-defined options.

Examples

See the `init-session` procedure in web\objects\web-util.p.

Previous topic: available-messages

Next topic: convert-datetime

convert-datetime

This function converts date and time between local and UTC (GMT) dates and times. Its return value is the `utc-offset` global variable.

Location

web\method\cgiutils.i

Parameters

INPUT p_conversion AS CHARACTER

The type of conversion to perform. The valid values are:

- **"UTC"** — Converts date and time from local to UTC time.
- **"LOCAL"** — Converts date and time from UTC to local time.
- **"NORMALIZE"** — Normalize date and time so the value of time is legal between zero and the number of seconds per day.

INPUT p_idate AS DATE

The date to convert. Uses the DATE data type.

INPUT p_itime AS INTEGER

The time to convert, expressed as seconds since midnight.

OUTPUT p_odate AS DATE

The converted date.

OUTPUT p_otime AS INTEGER

The converted time.

Returns

CHARACTER

Notes

The conversions between local and UTC time are also normalized so that the value of time is between zero and the number of seconds per day.

Examples

See the `format-datetime` function in web\method\cgiutils.i.

Previous topic: check-agent-mode
Next topic: delete-cookie

delete-cookie

This function deletes a persistent state cookie. This functions uses the `set-cookie` function to create a blank cookie with the same name, path, and an already expired date. This is only useful if the user's Web browser supports persistent state cookies and has cookies enabled.

Location

web\method\cookies.i

Parameters

INPUT p_name AS CHARACTER

The name of the cookie to delete.

INPUT p_path AS CHARACTER

The URL path to which the cookie should apply. If the `Unknown value (?)` is specified, the value of the `DefaultCookiePath` configuration option is used. If not set, the value of the `AppURL` global variable is used.

INPUT p_domain AS CHARACTER

The domain to which the cookie should apply. If the `Unknown value (?)` is specified, the value of the `DefaultCookieDomain` configuration option is used. If not set, the domain option is not set on the cookie and the current hostname of the Web server is used by the Web browser.

Returns

CHARACTER

Notes

- This function must be executed before the `output-content-type` function.
- Before using cookies, you should read the standards published for the various Web servers.

Examples

See the `continue-processing` procedure in web\objects\stateaware.p.

See also

set-attribute-list

get-cookie

delete-cookie

Previous topic: convert-datetime
Next topic: format-datetime

format-datetime

This function returns a date and time string formatted according to Internet standards.

Location

web\method\cgiutils.i

Parameters

INPUT p_format AS CHARACTER

The type of format to apply. The valid values are:

- **Cookie** — Formats a date and time string for use with cookies, for example when specifying an expiration date with the set-cookie function.
- **HTTP** — Formats a date and time string for use in header information.

INPUT p_date AS DATE

The date to format.

INPUT p_time AS INTEGER

The time expressed as seconds since midnight.

INPUT p_options AS CHARACTER

Specifies how to process the date and time. Valid values are:

- **Local** — The date and time are local values. The function converts them to UTC values before formatting them.
- **UTC** — The date and time are already in UTC time. The function normalizes them to ensure the value of time is between zero and the number of seconds in one day.

Returns

CHARACTER

Notes

None

Examples

See the `set-cookie` function in web\method\cookies.i.

See also

convert-datetime

set-attribute-list

Previous topic: delete-cookie
Next topic: getAttribute

getAttribute

This procedure accepts the name of a supported attribute (for example, `Type`, `Version`, `Web-State`, `Web-Timeout`, `Web-Timeout-Handler`, or `Web-Time-Remaining`) and returns its value as a character string (in `RETURN-VALUE`).

Location

web2\admweb.p

Parameters

INPUT p_attr-name AS CHARACTER

Name of the attribute.

Returns

CHARACTER

Notes

In addition to the attributes named above, other names can be used provided the `special-get-attribute` procedure exists in the target procedure to handle them.

Examples

```
\* Show the Type and Web.State of this procedure. *\
RUN getAttribute IN THIS-PROCEDURE ('Type':U).
{&OUT} '<BR>Type = ' RETURN-VALUE.

RUN getAttribute IN THIS-PROCEDURE ('Web-State':U).
{&OUT} '<BR>Web-State = ' RETURN-VALUE.
```

Previous topic: format-datetime

Next topic: get-binary-data

get-binary-data

This function returns a `MEMPTR` to the contents of a file posted from a Web page by a multipart/form-data form. The function uses the `GET-BINARY-DATA` method of the `WEB-CONTEXT` system handle.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The name of a field on the form. The field's type must be `file`.

Returns

MEMPTR

Note

If the uploaded file exceeds the limit set in the `binaryUploadMaxSize` property, the function returns the `Unknown value (?)` instead of a `MEMPTR`.

Example:

```
/* This example shows how to retrieve the data from a file passed on a given
request, and save it as a file with the same name to the WebSpeed's working
directory. Note that 'filename' is the name of the field in the form posted. */

DEFINE VAR mFile AS MEMPTR NO-UNDO.
DEFINE VAR cfile AS CHAR NO-UNDO.

/* 'filename' refers to the name of the field in the form. get-binary-data
returns the contents of the file associated with the form field named 'filename'. */
ASSIGN mFile = get-binary-data("filename").
 IF mFile <> ? THEN DO:
   /* if we got a valid pointer, save data to file. The value of the field
'filename' is the file name posted */
    ASSIGN cfile = get-value("filename").

    COPY-LOB FROM mFile TO FILE cFile NO-CONVERT.

 END.
```

Previous topic: getAttribute

Next topic: get-cgi

get-cgi

This function returns any CGI or other environment variables specific to a Web request. The function returns a blank value if an invalid variable name is supplied.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The variable name. If the `Unknown value (?)` is specified, the return value is a list of variables.

Returns

CHARACTER

Notes

Most standard CGI and a number of HTTP environment variables are available as global variables. This function is only needed to access environment variables not defined in web\method\cgidefs.i.

Examples

See the `init-cgi` procedure in web\objects\web-util.p.

See also

get-field

get-cgi-long

get-cookie

Previous topic: get-binary-data
Next topic: get-cgi-long

get-cgi-long

This function returns any CGI or other environment variables specific to a Web request as a `LONGCHAR` data type. This function serves as a wrapper for the `GET-CGI-LONGCHAR-VALUE` method of the `WEB-CONTEXT` system handle.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The variable name. If the `Unknown value (?)` is specified, the return value is a list of variables.

Returns

LONGCHAR

Notes

- Most standard CGI and a number of HTTP environment variables are available as global variables. This function is only needed to access environment variables not defined in web\method\cgidefs.i.
- INPUT/OUTPUT operations are not allowed with LONGCHAR variables. For example, the following code would fail:

```
{&OUT} vpath.
```

Examples

```
DEFINE VARIABLE vpath AS LONGCHAR.
vpath = get-cgi-long("PATH-INFO":U).
```

See also:

get-field

get-cgi

get-cookie

Previous topic: get-cgi
Next topic: get-config

get-config

This function returns the value of the specified configuration option for the WebSpeed service under which the agent is running.

Location

web\objects\web-util.p

Parameters

INPUT cVarName AS CHARACTER

The name of the configuration option from the `ubroker.WS` section of the ubroker.properties file.

Returns

CHARACTER

Notes

None

Examples

```
cfg-cookiepath = get-config("defaultCookiePath":U)
```

Previous topic: get-cgi-long
Next topic: get-cookie

get-cookie

This function returns the value of the specified cookie. If no cookie is found, the function returns a blank value. The return values are returned as a delimited string. The delimiter is the value of `SelDelim` in web \method\cgidefs.i, by default, a comma.

Location

web\method\cookies.i

Parameters

INPUT p_name AS CHARACTER

The cookie name. If the `Unknown value (?)` is specified, the function returns a list of all cookie names.

Returns

CHARACTER

Notes

- Requires a browser supporting persistent state cookies. The browser must also have cookies enabled or they will not be sent.
- Before using cookies, you should read the standards published for the various Web servers.

Examples

See the `init-request` procedure in web\objects\stateaware.p.

See also

set-attribute-list

delete-cookie

delete-cookie

get-field

get-cgi

Previous topic: get-config
Next topic: get-field

get-field

This function returns the value of an HTML form input field or named argument. If an invalid name is passed, the function returns a blank value. or returns a blank otherwise.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The field name. If the `Unknown value (?)` is specified, the function returns a list of all field names.

Returns

CHARACTER

Notes

None

Examples

See the get-value function in web\method\cgiutils.i.

See also

get-user-field

get-cgi

get-cookie

Previous topic: get-cookie

Next topic: get-long-value

get-long-value

This function returns the first available value of the specified user field, form field, or cookie. The function returns a blank value if passed an invalid name.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The name of the user field, form field, or cookie.

Returns

LONGCHAR

Notes

User fields are cleared with each new Web request.

Examples

```
DEFINE VARIABLE cAddress AS LONGCHAR.
cAddress = get-long-value("Address":U).
```

Previous topic: get-field
Next topic: get-message-groups

get-message-groups

This function returns a comma-delimited list of all the different message groups that have queued messages.

Location

web\method\message.i

Parameters

None

Returns

CHARACTER

Notes

All queued messages are output after the Web object specified in a URL returns. As such, messages do not remain queued between requests.

Examples

```
get-message-groups()
```

See also

available-messages

get-messages

output-messages

queue-message

Previous topic: get-long-value
Next topic: get-messages

get-messages

This function returns any messages queued for the specified message group or any message group if the specified group is `Unknown value (?)`. Messages are returned delimited by a linefeed character (`~n`).

Location

web\method\message.i

Parameters

INPUT p_grp AS CHARACTER

A known message group into which messages may have been queued with the `queue-message` function. The `Unknown value (?)` indicates all message groups.

INPUT p_delete AS LOGICAL

Indicates if the message queue should be emptied. If `TRUE` is specified, the messages are removed from the queue. If `FALSE` is specified, the messages remain queued.

Returns

CHARACTER

Notes

All queued messages are output after the Web object specified in a URL returns. As such, messages do not remain queued between requests.

Examples

See the `output-messages` procedure in web\method\message.i.

See also

available-messages

get-message-groups

output-messages

queue-message

Previous topic: get-message-groups
Next topic: get-transaction-state

get-transaction-state

This procedure returns the current database transaction state for the agent as the procedure's `RETURN-VALUE`. The six possible values are: `'NONE'`, `'ACTIVE'`, `'START-PENDING'`, `'UNDO-PENDING'`, `'RETRY-PENDING'`, and `'COMMIT-PENDING'`.

Location

web\objects\web-util.p

Parameters

None

Returns

CHARACTER

Notes

- Valid entries for the transaction state are checked in `set-transaction-state`. The value returned is one of the six valid values: `'NONE'`, `'ACTIVE'`, `'START-PENDING'`, `'UNDO-PENDING'`, `'RETRY-PENDING'`, and `'COMMIT-PENDING'`. This is true even if you use the shorthand abbreviations for the states (for example, `'START'` or `'UNDO'`).
- When the state is `'START-PENDING'`, it means that there is no active database transaction for the agent. `'NONE'` and `'START-PENDING'` only occur when there is no transaction.
- `'UNDO-PENDING'`, `'RETRY-PENDING'`, and `'COMMIT-PENDING'` all imply that there is an `'ACTIVE'` transaction. These values can only be set in Web requests where the transaction state is `'ACTIVE'`. None of these actions has any impact until the current request is complete.
- If the state is `'UNDO-PENDING'` or `'RETRY-PENDING'`, do not set any undo-able variables or tables, unless their values after the current request completes is unimportant. All changes to these variables and tables are lost at the end of the current Web request, when the current database transaction is undone.

Examples

The first example checks the transaction state before a database transaction starts. In this case, the Web object starts a new transaction whether or not one already exists, as shown:

```
RUN get-transaction-state IN web-utilities-hdl.
IF RETURN-VALUE eq 'NONE':U THEN
  RUN set-transaction-state IN web-utilities-hdl ('START':U).
ELSE IF RETURN-VALUE eq 'ACTIVE':U THEN
  RUN set-transaction-state IN web-utilities-hdl ('RETRY':U).
```

The second example shows how to avoid running two Web objects, where both rely on database transactions. If there already is a transaction, an error message is generated. This is an important test because all requests to a WebSpeed agent run within the same transaction. A single **UNDO** button will undo the work done by all Web objects running on the same agent, as shown:

```
\* Report an error message if there is already an ACTIVE transaction. *\
RUN get-transaction-state IN web-utilities-hdl.
IF RETURN-VALUE eq 'UNDO':U THEN DO:
  \* OK to turn this object into a state-aware object
     using a DB TRANSACTION. *\
  setWebState(20.0).
  RUN set-transaction-state IN web-utilities-hdl ('START':U).
END.
ELSE DO:
  \* Don't allow this object to start up
     because a DB TRANSACTION is active. *\
  RUN HtmlError IN web-utilities-hdl
  ('Close the current database transaction before starting this web object.').
  RETURN.
END.
```

See also

set-transaction-state

Previous topic: get-messages
Next topic: get-user-field

get-user-field

This function returns the value of the specified user settable field that was set with the `set-user-field` function.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The user field name. If the `Unknown value (?)` is specified, the list of user fields is returned.

Returns

CHARACTER

Notes

- Using `set-user-field`, one Web object can make data available to other Web objects. A Web object can access that information with the `get-user-field` or `get-value` functions.
- User fields are cleared with each new Web request.

Examples

```
cAddress = get-user-field("Address":U)
```

See also

set-user-field

get-field

hidden-field-list

url-field-list

url-format

Previous topic: get-transaction-state
Next topic: get-value

get-value

This function returns the first available value of the specified user field, form field, or cookie. If an invalid name is passed, the function returns a blank value.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The user field, form field, or cookie. If the `Unknown value (?)` is specified, a comma-delimited list of all user fields, form fields, and cookies is returned.

Returns

CHARACTER

Notes

- Using `set-user-field`, one Web object can make data available to other Web objects. A Web object can access that information with the `get-user-field` or `get-value` functions.
- User fields are cleared with each new Web request.

Examples

See the `hidden-field-list` function in web\method\cgiutils.i.

See also

set-user-field

get-user-field

get-field

1

Previous topic: get-user-field
Next topic: hidden-field

hidden-field

This function returns an HTML hidden field with the name and value encoded with HTML entities. The output is in the following format:

```
<INPUT TYPE="HIDDEN" NAME="encoded-field-name" VALUE="encoded-field-value">
```

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

Name of field to set in the `NAME` attribute. Any legal HTML name can be used.

INPUT p_value AS CHARACTER

Value to set in the `VALUE` attribute.

Returns

CHARACTER

Notes

None

Examples

See the `hidden-field-list` function in web\method\cgiutils.i.

See also

hidden-field-list

html-encode

Previous topic: get-value
Next topic: hidden-field-list

hidden-field-list

This function accepts a list of fields and returns newline-delimited (~n) list of HTML tags formatted to hide those fields. The output is in the following format:

```
<INPUT TYPE="HIDDEN" NAME="encoded-field-name" VALUE="encoded-field-value">
```

Location

web\method\cgiutils.i

Parameters

INPUT p_name-list AS CHARACTER

Character expression containing a comma-separated list of field names available through the `get-value` function.

Returns

CHARACTER

Notes

The value associated with each field name is determined by calling the `get-value` function.

Examples

```
cHiddenFields = hidden-field-list("user-name,login-company,password":U)
```

See also

hidden-field

get-value

url-field-list

url-format

Previous topic: hidden-field
Next topic: html-encode

html-encode

This function converts various characters that can be misinterpreted as HTML formatting to their HTML entity representation. The characters <, >, & and " (double quote) are converted to `<`, `>`, `&`, and `&quot;` respectively.

Location

web\method\cgiutils.i

Parameters

INPUT p_in AS CHARACTER

The character string to encode.

Returns

CHARACTER

Notes

This function cannot be called more than once on a string. This is because ampersands are replaced with `&` and recursive calls will fail.

Examples

```
PROCEDURE displayFields:
\*------------------------------------------------------------------------
Purpose: Override standard ADM method
------------------------------------------------------------------------*\
RUN SUPER.
\* The comment field may contain <, >, quotes and ampersands. To be safe, convert
these characters. *\
IF AVAILABLE Customer THEN
  Customer.Comments:SCREEN-VALUE IN FRAME {&FRAME-NAME} =
  html-encode(Customer.Comments).
END PROCEDURE.
```

Previous topic: hidden-field-list

Next topic: HtmlError

HtmlError

This procedure outputs an HTML-formatted error message, including the MIME Content-Type header if required.

Location

web\objects\cgiutils.i

Parameters

INPUT p_error AS CHARACTER

The text string to convert.

Notes

- This procedure assumes it is generating the entire return page. As such, it outputs the HTML `<HEAD>` and `<BODY>` sections.
- The `HelpAddress` global variable can be set to a `mailto` or other URL indicating who should be contacted in the event there are problems with the application.

Examples

See the `continue-processing` procedure in web\objects\stateaware.p and the `adm-output-fields` procedure in `web\method\html-map.i`.

Previous topic: html-encode
Next topic: output-content-type

output-content-type

This function sets and outputs the HTTP Content-Type header with the specified value followed by a blank line. If this function is called more than once per Web request, no output is generated after the first call. The function returns `TRUE` if a header was output and returns `FALSE` otherwise.

Location

web\method\cgiutils.i

Parameters

INPUT p_type AS CHARACTER

The MIME content type. If the input value is blank, then no Content-Type header is output.

Returns

LOGICAL

Notes

- A Content-type header is mandatory for any content sent to the Web output stream. Otherwise, the Web server will probably return an error to the browser. The Web object templates already execute this function with a default Content-type of `text\html`.
- Since this function outputs a blank line, it must be executed after `output-http-header`, `set-cookie`, `delete-cookie`, or other HTTP headers.

Examples

Outputs the text\plain MIME type with a charset modifier followed by the name of the browser. The tilde character (~) is used to escape the semi-colon to ensure that the semicolon is taken literally, as shown:

```
output-content-type("text\plain~; charset=iso-8859-1":U).
{&OUT}
'Your web browser is ' HTTP_USER_AGENT SKIP
{&END}
```

If the Web server supports Server Side Includes (SSI) and is configured to utilize it, the MIME type `application\x-server-parsed-html` can be used to have the WebSpeed output further parsed by the Web server before being sent to the Web browser, as shown:

```
output-content-type("application\x-server-parsed-html":U).
{&OUT}
'<!--#include virtual="\header.html" -->' SKIP
. . .
'<!--#include virtual="\footer.html" -->' SKIP
{&END}
```

See also

output-http-header

Previous topic: HtmlError
Next topic: outputHeader

outputHeader

This procedure outputs a MIME header and any cookie information needed by the Web object.

Location

web2\template\html-map.w

web2\template\wrap-cgi.w

Parameters

None

Notes

- Always run `outputHeader` before generating any HTML code. All new cookies must be created prior to the call to `outputContentType` that is embedded in `outputHeader`.
- If one Web object is going to run another Web object, remember that only the Web object that actually creates the HTML page should run `outputHeader`.
- Modify `outtputHeader` when you want a Web object to be state-aware or when you want to add special cookie information in your application. Instructions for doing both these tasks are provided in the template for this procedure. For state-aware Web objects, this procedure is a good place to set the `webState` and `webTimeout` attributes, as follows:

```
PROCEDURE outputHeader :
\*------------------------------------------------------------------
Purpose: Output the MIME header, and any "cookie" information
------------------------------------------------------------------*\
\* Make this a State-Aware web object with a 15 minutes timeout period *\
setWebState(5.0).
\* Create a standard HTML page. *\
outputContentType("text\html":U).
END PROCEDURE.
```

Examples

See the `process-web-request` function in web2\template\html-map.w.

See also

output-content-type

output-http-header

set-attribute-list

Previous topic: output-content-type
Next topic: output-http-header

output-http-header

This function outputs the specified HTTP header with an associated value followed by a carriage return and linefeed. If the header name is blank, then the value and carriage return\linefeed pair are still output.

Location

web\method\cgiutils.i

Parameters

INPUT p_header AS CHARACTER

The HTTP Header name.

INPUT p_value AS CHARACTER

The header value.

Returns

CHARACTER

Notes

This procedure must be executed before the `output-content-type` function.

Examples

This should cause the Web server to do an internal redirection for the specified URL in the location header, as shown:

```
output-http-header("Status":U, "302
Redirect":U).output-http-header("Location":U,
"\cgi-bin\demo.sh\restart.w":U).output-http-header("", "").
```

See also

output-content-type

Previous topic: outputHeader
Next topic: output-messages

output-messages

This function outputs all messages queued for a specified message group or all message groups in HTML. If no prior output has been sent for the current Web request, an entire HTML page is generated with appropriate Content-type header, head, and body sections. Otherwise, the messages are output in the format of an HTML unnumbered bulleted list. The returned value is the number of messages output.

Location

web\method\message.i

Parameters

INPUT p_option AS CHARACTER

Specifies the messages to output. The valid values are as follows:

- **"page"** — Output all queued messages as a single HTML page.
- **"all"** — Output all queued messages, but no HTML title and H1 are output. This option is suitable for dumping all messages out from an application.
- **"group"** — Output just the messages in a single group as specified with `queue-message` function.

INPUT p_grp AS CHARACTER

If using the "group" setting for p_option, the name of message group from which to output messages. If the `Unknown value (?)` is specified, outputs all messages.

INPUT p_message AS CHARACTER

An optional message heading. With the "page" option, this text is displayed in the HTML title and H1 sections. With the "all" option, or "group" option, the text is displayed before the messages are output as a heading.

Returns

INTEGER

Notes

- All queued messages are output after the Web object specified in the URL returns. As such, messages do not remain queued between requests.
- If there are no messages to be output matching the requested input parameters, then no heading or other output is generated so it is not necessary to see if there are any messages to output before calling this function.

Examples

See the `PrintVars` procedure in web\support\printval.p.

See also

available-messages

get-message-groups

get-messages

queue-message

Previous topic: output-http-header
Next topic: process-web-request

process-web-request

This is the primary procedure for handling Web requests. Web objects run this procedure to respond to form input, as well as to generate HTML pages.

Location

web\template\html-map.w

web\template\wrap-cgi.w

Parameters

None

Notes

- Both the HTML Mapping Procedure and CGI Wrapper Procedure templates provide suggested code for this routine (the code differs between the two templates). The specific requirements of your Web objects might require modification of that code.
- When the Main Code Block of a Web object runs, it calls `process-web-request`. If a Web object is state-aware, subsequent calls to the Web object from the Web browser also run `process-web-request`.
- This procedure has four basic functions:
  - It determines the context in which it is running. This can be done by checking `REQUEST_METHOD` or by explicitly using `get-field` or `get-cookie` to determine how the Web object was called.
  - Once the context is determined, `process-web-request` calls `outputHeader` (unless modified otherwise). This sets the MIME header and any cookies on the HTML page that is being returned.
  - Optionally, `process-web-request` will read form input from the Web request and populate local fields and variables. This will only be done if the context found in the first step indicates that data is being submitted with the form.
  - In all cases, a Web object will cause some sort of response to be returned to the WEBSTREAM. This can be an HTML form generated by the Web object itself, or a response generated by a second Web object called by the current Web object.
- Note that `outputHeader`**must** be run from `process-web-request` prior to generating any HTML code. It is also important that all new cookies are created prior to the call to `outputHeader`.
- If one Web object is going to run another Web object, it is also important to remember that they should not both run `outputHeader`. Only the Web object that is actually creating the HTML page should run `outputHeader`.

Examples

```
RUN process-web-request.
```

See also

run-web-object

Previous topic: output-messages

Next topic: queue-message

queue-message

This function queues a message for later output by either the `output-messages` or `get-message` functions. A message group is user-defined and can be used by different parts of an application to queue related messages together.

Location

web\method\message.i

Parameters

INPUT p_grp AS CHARACTER

The message group into which messages will be queued.

INPUT p_message AS CHARACTER

The message text.

Returns

INTEGER

Notes

- The message text may contain HTML tags. However, if the message text might have characters that could be incorrectly interpreted as HTML tags, use the `html-encode` function to ensure such characters are encoded so this does not happen.
- All queued messages are output after the Web object specified in a URL returns. As such, messages do not remain queued between requests.

Examples

```
\* Queue up two messages in the "validate" group *\
queue-message("validate", html-encode("invalid zipcode <" + zip-code + "> was
entered")).
queue-message("validate", "City name " + html-encode(customer.city) + " is
unknown").
. . .
\* Output all messages in the validate group with a heading *\
output-messages("group", "validate", "Validation Errors").
. . .
\* Output any remaining messages that are queued *\
output-messages("all", ?, "Other Messages").
```

See also

available-messages

get-message-groups

queue-message

output-messages

Previous topic: process-web-request
Next topic: run-web-object

run-web-object

This procedure attempts to run a procedure as a WebSpeed Web object. By default, all Web objects run with `run-web-object` are run as persistent procedures. However, if the `STATE-AWARE-ENABLED` environment variable has a value other than "YES", WebSpeed does not load web\objects\stateaware.p and does not run the procedure as a persistent procedure.

Location

web\objects\web-util.p

Parameters

INPUT pcFilename AS CHARACTER

The filename of the Web object to run.

Notes

The `filename` for the Web object must exist in a directory listed in the `PROPATH` for the WebSpeed agent. If only compiled r-code (*.r) exists it will be run, even if the requested file name explicitly matches the source file (*.w).

Examples

```
\* Run another web object. Make sure REQUEST_METHOD is appropriate for that
object *\
REQUEST_METHOD = 'GET':U.
RUN run-web-object IN web-utilities-hdl ('success.w':U).
```

Previous topic: queue-message

Next topic: set-attribute-list

set-attribute-list

This procedure accepts the value of the complete object attribute list and runs procedures to set individual attributes.

Location

web2\admweb.p

Parameters

INPUT p-attr-list AS CHARACTER

A comma-delimited attribute list with the format `name=value`. Typical attributes are `web-timeout`, `web-state`, and `web-timeout-handler`. Other names can be used provided the `getAttribute` procedure exists in the target procedure to handle them.

Notes

Not all attributes are settable. Those which are a part of an event such as enable\disable (which set `ENABLED` on\off) or hide\view (which set `HIDDEN` on\off) can be queried through `getAttribute`, but are read-only.

Examples

```
RUN set-attribute-list ("web-state=persistent, web-timeout=60").
```

See also

available-messages

Previous topic: run-web-object
Next topic: set-cookie

set-cookie

This function sets a persistent state cookie in the Web browser by outputting an HTTP `Set-Cookie` header with specified options. The Web browser on future requests sends the name and associated value of the cookie automatically. This is especially useful for preserving state information between requests.

Location

web\method\cookies.i

Parameters

INPUT p_name AS CHARACTER

The name of the cookie.

INPUT p_value AS CHARACTER

The value of the cookie.

INPUT p_date AS DATE

The optional expiration date (local). If the `Unknown value (?)` is specified, the cookie expires when the browser session ends.

INPUT p_time AS INTEGER

The optional expiration time (local) as a number of seconds since midnight. This parameter only has meaning if p_date is not the `Unknown value (?)`. If you do not want to set a specific time, specify the `Unknown value (?)`.

INPUT p_path AS CHARACTER

The URL path to which the cookie should apply. If `Unknown value (?)` is specified, the value of the `DefaultCookiePath` configuration option is used or, if that is not set, the value of the `AppURL` global variable.

INPUT p_domain AS CHARACTER

The optional domain to which the cookie should apply. If `Unknown value (?)` is specified, value of the `DefaultCookieDomain` configuration option is used or, if that is not set, the domain option is not set on the cookie. In that case, the current hostname of the Web server is used by the Web browser.

INPUT p_options AS CHARACTER

A comma-delimited list of options. The valid values are as follows:

- **Secure** — If specified, the Web browser only sends the Cookie back when on a secure (SSL) connection (using HTTPS).
- **Local** — (Default) Assume date and time are based on local time and need conversion to UTC.
- **UTC** — Assume date and time are UTC based eliminating any conversion that would otherwise be required from local time to UTC time.

Returns

CHARACTER

Notes

- If the p_time expression is a very large or small number (greater or less than the number of seconds in a day), it is normalized to fit within a day. In this case, the expiration date is incremented or decremented as appropriate.

- Because of possible differences between the user's machine and your server's clocks, cookies based on functions like TIME might not expire exactly when you expect them to expire.
- Requires a browser supporting persistent state cookies. Other browsers should ignore the HTTP `Set-Cookie` header.
- Cookies must be set within the HTML header. Therefore, for all practical purposes, all cookies must be defined prior to the first call to the `output-content-type` function.
- Before using cookies, you should read the standards published for the various Web servers.

Examples

```
\* Sets cust-num = 23 *\
set-cookie("cust-num":U, "23":U, ?, ?, ?, ?, ?).

\* Set USER-ID valid in or below the current web object with an expiration time
of one week from today. *\
set-cookie("UserId":U, USER-ID, TODAY + 7, ?, ?, ?, ?).
```

See also

output-content-type

delete-cookie

get-cookie

delete-cookie

Previous topic: set-attribute-list
Next topic: set-transaction-state

set-transaction-state

This procedure changes the current database transaction state for the WebSpeed agent. The database transaction state can have the following values:

- `NONE` — The agent is not in a database transaction.
- `START-PENDING` — The agent is currently not in a database transaction, but will start the next request cycle in a database transaction (after the current request completes).
- `ACTIVE` — The agent is currently in a database transaction.
- `UNDO-PENDING` — The agent is in a database transaction that will be undone before the next request cycle (after the current request completes).
- `COMMIT-PENDING` — The agent is in a database transaction that will be committed before the next request cycle (after the current request completes).
- `RETRY-PENDING` — The agent is in a database transaction that will be undone before the next request cycle, when a new transaction will be started. (This is equivalent to `UNDO` and `START` before the next request cycle.)

Location

web\objects\web-util.p

Parameters

INPUT pState AS CHARACTER

The database transaction state to set. Only four of the six database transaction states can be set programatically: START-PENDING, UNDO-PENDING, RETRY-PENDING, and COMMIT-PENDING.

Notes

- You can omit -PENDING when typing state values in your code. For example, START is equivalent to START-PENDING.
- You can only set the transaction state to START-PENDING if the current state value is NONE. You can only set the transaction state to UNDO-PENDING', RETRY-PENDING, and COMMIT-PENDING if the current state value is ACTIVE. Trying to set the transaction state in any other case results in an error.
- If you attempt to start a transaction, and there are no state-aware Web objects created or in existence at the end of the Web request, then the transaction state reverts to NONE for the next request.
- States of ACTIVE, UNDO-PENDING, RETRY-PENDING and COMMIT-PENDING all indicate that a database transaction is active.
- When you set the transaction state, the new state does not take affect until the start of the next request cycle. It is therefore good practice to set the value near the bottom of your process-web-request, where you are certain no changes to variables or tables will occur in the current request that you might want undone in a subsequent request.
- If the transaction state is UNDO-PENDING or RETRY-PENDING, do not set any undo-able variables or tables, unless their values after the current request completes is unimportant. All changes to these variables and tables are lost at the end of the current Web request, when the current database transaction is undone. Similarly, if you look at the value of these variables and fields during the next request, they will show the old values for the duration of the Web request.

Examples

```
\* Get the submitAction field (set on SUBMIT buttons) and reset the
transaction-state.
Note that on a COMMIT or UNDO, we want to return to the menu page. The current
page will be returned only in the RETRY case. *\
DEFINE VARIABLE c_action AS CHARACTER NO-UNDO.
c_action = get-field('submitAction':U).
CASE c_action:
WHEN 'UNDO':U OR WHEN 'COMMIT':U THEN DO:
RUN set-transaction-state IN web-utilities-hdl (c_action).
RUN run-web-object IN web-utilities-hdl ('main.w':U).
RETURN.
END.
WHEN 'RETRY':U THEN DO:
RUN set-transaction-state IN web-utilities-hdl (c_action).
\* Output the MIME header. *\
```

```
RUN outputHeader.
END.
END CASE.
```

See also

get-transaction-state

Previous topic: set-cookie
Next topic: set-user-field

set-user-field

This function sets the value of a user-specified field. The new value can replace or be appended to the existing value. The function returns TRUE if the field was added and returns FALSE otherwise.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

The name of a user field to set. Almost any alphanumeric name can be used.

INPUT p_value AS CHARACTER

A character string with the value to which the field should be set.

Returns

LOGICAL

Notes

- If a field is set with a name that is the same as form input in the current Web request, certain functions will use the value of the user field instead of form input.
- The function queues a message if adding the field fails.
- Using `set-user-field`, one Web object can make data available to other Web objects. A Web object can access that information with the `get-user-field` or `get-value` functions.
- User fields are cleared with each new Web request.

Examples

```
cEmail = user-name + "@":U + login-company + ".com":U
set-user-field("email-address",cEmail)
```

See also

get-user-field

get-field

hidden-field-list

url-field-list

url-format

Previous topic: set-transaction-state
Next topic: setWebState

setWebState

This procedure sets `web-state` for the current Web object. When web-state changes, the appropriate cookie information is also set.

Location

web2\admweb.p

Parameters

INPUT pdWebTimeout AS DECIMAL

The timeout period, in minutes, for the Web object. Specify the time to one decimal place. For example, enter `0.0` for no timeout period or `15.0` to signify fifteen minutes.

Returns

LOGICAL

Notes

- This function is called from `outputHeader` in a Web object to change the `web-state` and `web-timeout` attributes. In addition, changing these attributes sets the WSEU (WebSpeed Exclusive User) cookie that instructs WebSpeed to lock the agent process (and therefore dedicate the agent to a particular end user).
- When `timeout-period` is set at a positive value, the `web-state` attribute for that object is set to `state-aware`, and the `web-timeout` attribute is set to the timeout period.
- A timeout period of 0 or less turns the `web-state` to `state-less` and resets the `web-timeout` attribute to 0.
- Note that `setWebState` must be called prior to `output-content-type` or it will not affect the header for the current HTML page.

Examples

```
PROCEDURE outputHeader:
\*---------------------------------------------------------------------
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. *\
setWebState(IF state EQ "complete":U THEN 0.0 ELSE 15.0).

\* Create a standard HTML page. *\
output-content-type("text\html":U).
END PROCEDURE.
```

See also

output-content-type

Previous topic: set-user-field
Next topic: url-decode

url-decode

This function decodes URL form input from either POSTs or GETs or encoded cookie values.

Location

web\method\cgiutils.i

Parameters

INPUT p_in AS CHARACTER

The encoded URL.

Returns

CHARACTER

Notes

None

Examples

See the `get-cookie` function in web\method\cookies.i.

See also

url-encode

Previous topic: setWebState
Next topic: url-encode

url-encode

This function encodes characters that can be misinterpreted in a URL (for example tilde ~ or percent %) as hexidecimal triplets. For instance, %7E is the tilde character and the percent character is %25.

Location

web\method\cgiutils.i

Parameters

INPUT p_value AS CHARACTER

The character expression to encode.

INPUT p_enctype AS CHARACTER

Either query, cookie, or default. The query option is suitable for encoding name and value pairs for use as arguments in a URL. The cookie option is suitable for encoding cookie contents. The default option is the default encoding as defined in RFC 1738, section 2.2.

Returns

CHARACTER

Notes

None

Examples

See the set-cookie function in web\method\cookies.i.

See also

url-decode

url-format

Previous topic: url-decode
Next topic: url-field

url-field

This function returns an encoded name\value pair suitable for use as an argument in a URL.

Location

web\method\cgiutils.i

Parameters

INPUT p_name AS CHARACTER

Name of field to set in the name attribute.

INPUT p_value AS CHARACTER

Value to set in the value attribute.

INPUT p_delim AS CHARACTER

Delimiter to return as a prefix. If the `Unknown value (?)` is specified, the default is `&` which is appropriate if the resulting URL is output to the browser.

Returns

CHARACTER

Notes

None

Examples

See the `urlJoinParams` function in web2\webrep.p.

See also

url-field-list

url-encode

Previous topic: url-encode
Next topic: url-field-list

url-field-list

This function returns a series of name\value pairs suitably encoded and delimited to send to a browser as a URL argument. If a blank or `Unknown value (?)` is passed in, the function returns a blank value.

Location

web\method\cgiutils.i

Parameters

INPUT p_name-list AS CHARACTER

A character expression containing a comma-delimited list of field names. These are available through the `get-value` function.

INPUT p_delim AS CHARACTER

The delimiter to use between name\value pairs. If the `Unknown value (?)` is specified, the default is `&` which is appropriate if the resulting URL is output to the browser.

Returns

CHARACTER

Notes

- Unlike the `url-field` function, no leading delimiter is added to the return string. This is so it can be easily appended to a `Unknown value (?)` in a URL.
- The value associated with each field name is determined by calling the `get-value` function.

Examples

See the `url-format` function in web\method\cgiutils.i.

See also

hidden-field-list

get-value

url-field

url-format

Previous topic: url-field
Next topic: url-format

url-format

This function formats a URL and arguments for all the specified fields that result in non-blank values. It returns a series of name\value pairs suitably encoded and delimited to send to a browser.

Location

web\method\cgiutils.i

Parameters

INPUT p_url AS CHARACTER

A URL without arguments or query strings. If a `Unknown value (?)` is specified for the URL, the value of the global variable `SelfURL` is used, which is the URL of the currently executing Web object without the host name and port number.

INPUT p_name-list AS CHARACTER

A character expression containing a comma-delimited list of field names available through the `get-value` function.

INPUT p_delim AS CHARACTER

The delimiter to use between name\value pairs. If the `Unknown value (?)` is specified, the default is `&` which is appropriate if the resulting URL is output to the browser.

Returns

CHARACTER

Notes

The value associated with each field name is determined by calling the `get-value` API function.

Examples

```
{&OUT} url-format("custdetail.html","name,custnum",?).
```

See also

url-field-list

get-value

url-field

hidden-field-list

Previous topic: url-field-list