Using Caché Server Pages (CSP)

Version 2018.1
2020-11-13

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Caché Server Pages (CSP)
Caché Version 2018.1 2020-11-13

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About this Book **1**

1 Introduction to Caché Server Pages **3**
1.1 CSP and Zen 3
1.2 Before You Start 4
1.2.1 Production Web Server and Caché-supplied Private Web Server 4
1.2.2 Configuring the Web Server and the CSP Gateway 4
1.2.3 What You Should Know 4
1.2.4 CSP Samples 5
1.2.5 CSP Documentation 5
1.3 Creating Your First CSP Page 6
1.3.1 Creating a Class-based CSP Page 6
1.3.2 Creating an HTML Tag-based CSP Page 7

2 CSP Architecture **9**
2.1 CSP Components: Web Server, CSP Gateway, CSP Server 9
2.1.1 What Each Component Does 10
2.1.2 Information Flow 10
2.1.3 Static Files 10
2.2 Web Server Configuration of a URL 11
2.2.1 Configuring a New URL on a Microsoft IIS Web Server 12
2.3 CSP Gateway Configuration 13
2.3.1 CSP Gateway Manager 13
2.3.2 Defining Server Access 13
2.3.3 Defining Application Access 14
2.3.4 CSP Gateway Parameters 15
2.4 CSP Application Settings 15
2.4.1 Enabling Application Access to %CSP Pages 15
2.4.2 Editing Web Application Settings 18
2.4.3 Defining a New Application 18

3 HTTP Requests in CSP **21**
3.1 CSP Runtime Environment 21
3.2 HTTP Request Processing 22
3.2.1 Web Server and the CSP Gateway 23
3.2.2 CSP Server 23
3.2.3 CSP Server Event Flow 23
3.2.4 CSP Server URL and Class Name Resolution 23
3.3 %CSP.Page Class 25
3.3.1 Page Method 25
3.3.2 %CSP.Page Class Parameters 25
3.3.3 Handling CSP Errors 26
3.4 %CSP.Request Object 27
3.4.1 URL Property 27
3.4.2 Data Property and URL Parameters 27
3.4.3 CgiEnvs Property and CGI Environment Variables 28
3.4.4 Cookies Property 28
3.4.5 MIME Data Property 28
3.5 %CSP.Response Object and the OnPreHTTP Method 28

3.5.1 Serving Cookies with the SetCookie Method 29
3.5.2 Serving Different Content Types 30

4 CSP Session Management 31
4.1 Sessions with CSP.Session 31
4.1.1 Session Creation 31
4.1.2 Session ID 32
4.1.3 Session Termination and Cleanup 32
4.1.4 Reserved CSP Parameters 32
4.2 %CSP.Session Object 35
4.2.1 User Session Data — Data Property 35
4.2.2 Setting User Session Data — Set Command 35
4.2.3 Retrieving User Session Data — Write Command 35
4.2.4 Deleting User Session Data — Kill Command 36
4.2.5 Session Timeout 36
4.2.6 Timeout Notification — OnTimeout Method 36
4.3 State Management 36
4.3.1 Tracking Data between Requests 36
4.3.2 Storing Data within a Page 37
4.3.3 Storing Data in Cookies 37
4.3.4 Storing Data in the Session — Data Property 38
4.3.5 Storing Data in the Database 38
4.3.6 Server Context Preservation — Preserve Property 38
4.4 Authentication and Encryption 38
4.4.1 Session Key 38
4.4.2 Encrypted URLs and CSPToken 38
4.4.3 Private Pages 39
4.4.4 Encoded URL Parameters 40
4.5 Authentication Sharing Strategies 41
4.5.1 Authentication Approaches 42
4.5.2 Authentication Architecture 44
4.5.3 Considerations in Choosing Your Strategy 45

5 Tag-based Development with CSP 49
5.1 CSP Compiler 49
5.1.1 Automatic and Manual Page Compilation 50
5.2 CSP Markup Language 50
5.2.1 CSP Page Language 51
5.2.2 Text 51
5.2.3 Compile-time Expressions and Code 51
5.2.4 Runtime Expressions 51
5.2.5 Runtime Code 53
5.2.6 Runtime Code ObjectScript Single Line 53
5.2.7 Server-Side Method 53
5.2.8 SQL <script> Tag 54
5.2.9 Controlling the Generated Class 54
5.3 Control Flow 55
5.3.1 <csp:if> Tag 55
5.3.2 <csp:while> Tag 55
5.3.3 <csp:loop> Tag: Numbered List Example 56
5.4 Escaping and Quoting HTTP Output 56
5.4.1 Escaping HTML with EscapeHTML 56

5.4.2 Escaping URL Parameters with EscapeURL 57
5.4.3 Escaping JavaScript with QuoteJS 57
5.5 Server-Side Methods 58
5.5.1 Caché and AJAX 58
5.5.2 Calling Server-side Methods via HTTP Submit 58
5.5.3 Calling Server-side Methods Using Hyperevents #server and #call 59
5.5.4 Tips for Using Server-Side Methods 62

6 Building Database Applications 67
6.1 Using Objects on a Page 67
6.1.1 Displaying Object Data in a Table 67
6.1.2 Displaying Object Data in a Form 68
6.1.3 Processing a Form Submit Request 69
6.1.4 <csp:object> tag 69
6.2 Binding Data to Forms 70
6.2.1 Binding to a Property 71
6.3 CSP Search Page with <csp:search> Tag 72
6.4 Enabling Logging in ISCLOG 73
6.4.1 Message Format 75

7 Localizing Text in a CSP Application 77
7.1 Localization Basics 77
7.2 $$$Text Macros 78
7.2.1 Argument Details 78
7.2.2 $$$Text at Compile Time 79
7.2.3 $$$Text at Runtime 80
7.3 Other Options for Displaying Localized Strings 80
7.3.1 %response.GetText Method 80
7.3.2 FormatText Method 80
7.3.3 $$$FormatText Macros 81
7.4 The MatchLanguage() Method 81

8 Developing Custom Tags 83
8.1 Rules and Actions 83
8.2 Tag Matching — match Attribute 85
8.3 Server-side Expressions and Code in Rule Actions 85
8.3.1 Runtime Expressions in Actions 85
8.3.2 Compile-time Expressions in Actions 86
8.3.3 <script> Tags in Actions 86
8.4 Server Document Object Model 86
8.4.1 Access Rule Attribute Values 87
8.5 Using <csr> Tags in Actions 88
8.5.1 <csr:default> Tag 88
8.5.2 <csr:children> Tag 88
8.5.3 <csr:section> Tag 88
8.6 Using <csr> Tags Outside Actions 89
8.6.1 <csr:class> Tag 89
8.6.2 <csr:property> Tag 89
8.6.3 <csr:description> Tag 90
8.6.4 <csr:attribute> Tag 90
8.7 Using Rule Classes 91
8.7.1 Structure of Generated Rule Classes 91

8.7.2 RenderStartTag Method 92
8.7.3 CompilerMethod[n]() Method 93
8.7.4 RenderEndTag Method 93
8.8 Using %CSP.Rule Methods 94
8.8.1 GetAttribute Method 94
8.8.2 QuoteAttribute Method 94
8.8.3 GetAttributesOrdered Method 95
8.8.4 IsDefined Method 95
8.8.5 InnerText Method 95
8.8.6 AddChildElement Method 95
8.8.7 SetAttribute Method 96
8.8.8 OnMatch Method 96
8.9 Using <csr> %CSP.AbstractAtom Write Methods 96
8.9.1 WriteText Method 97
8.9.2 WriteCSPText Method 97
8.9.3 WriteExpressionText Method 97
8.9.4 WriteServer Method 97
8.9.5 WriteCSPServer Method 98
8.10 Using <csr> %cspQuote Methods 98
8.10.1 Quote Method 98
8.10.2 QuoteCSP Method 98
8.11 Creating a <grid> Tag to Display a Table 99
8.11.1 Grid Rule Definition 99
8.11.2 Generated Grid Class 99
8.11.3 Using the Grid Rule 100
8.11.4 Grid Rule Displayed Page 101

Appendix A: CSP Error Notes 103

Appendix B: Localization and Tag-Based Development 109
B.1 Introduction 109
B.2 Localization Tags at Runtime 109
B.2.1 Default Language 110
B.2.2 Default Domain 110
B.2.3 Message Arguments 110
B.2.4 Button Text 110
B.3 Localization Tags at Compile Time 111

Appendix C: Frequently Asked Questions About CSP 113

List of Figures

Figure 2–1: CSP Components 9
Figure 3–1: CSP Architecture 21
Figure 3–2: HTTP Event Flow 22

List of Tables

Table 3–1: URL Components 24
Table 4–1: Reserved CSP Parameters 33
Table 6–1: Effects of cspbind Attribute 71
Table 6–2: HTML Input Elements Supported by cspbind 72
Table 6–3: <csp:search> Tag Attributes 72
Table 6–4: ISCLOG Fields 74
Table 8–1: Examples of Tag Matching 85
Table I–1: CSP Error Codes, Error Messages, and When Reported 103

About this Book

This book is written for web application developers.

This book describes how to create Caché Server Pages (CSP) for rapid application development.

Chapters are:

- Introduction to Caché Server Pages
- CSP Architecture
- HTTP Requests
- CSP Session Management
- Tag-Based Development with CSP
- Building Database Applications
- Localizing Text in a CSP Application
- Developing Custom Tags

Appendixes are:

- CSP Error Notes
- Localization and Tag-Based Development
- Frequently Asked Questions About CSP

A detailed Table of Contents.

1

Introduction to Caché Server Pages

Caché Server Pages (CSP) is both an architecture and toolset used to build an interactive CSP application. CSP technology allows you to build and deploy high-performance, highly scalable web applications. CSP lets you dynamically generate web pages, typically using data from a Caché database. "Dynamically" means that the same page may deliver different content each time it is requested from recently changed data sources.

CSP is versatile. It can

- Display inventory data that changes minute by minute.
- Support web communities with thousands of active users.
- Personalize pages based on user information stored in the Caché database.
- Customize pages based on the user data to different users, depending on their requirements and their security permissions.
- Serve HTML, XML, images, or other binary or textual data.
- Deliver fast performance, because it is tightly coupled to the high-performance Caché database.

CSP is well-suited for database applications. In addition to providing rapid access to the built-in Caché database, it provides a number of features essential for Web-based database applications including

- Session management
- Page authentication
- Ability to perform interactive database operations from in a web page.

CSP supports two styles of web development.

- To develop applications using classes, CSP provides an object framework.
- To develop applications using HTML files, CSP provides an HTML-based markup language that allows the inclusion of objects and server-side scripts within web pages. You can combine these two techniques within an application for maximum flexibility.

1.1 CSP and Zen

Zen is an application framework for quickly creating data-rich web applications that is included with Caché and Ensemble. If you are building a new web-based application or enhancing an existing CSP-based application, look at the Zen framework in Using Zen.

Note: To run Zen-based applications, it is necessary that you enable the **Serve Files** option and properly configure your web server. See the section Static Files in this book for more information.

1.2 Before You Start

This book assumes that you have a web server and Caché installed. This section describes what you need to do to be ready to create CSP applications.

1.2.1 Production Web Server and Caché-supplied Private Web Server

Caché supplies a minimal web server that runs the Management Portal, sometimes called the *private web server*. The private web server can also display the supplied CSP samples and run CSP pages. It cannot run robust CSP applications in a production environment. For that, you need to install a separate, full installation of a supported web server, such as Apache web server, Microsoft's IIS web server, or a Sun web server.

The private web server is based on a minimal build of the Apache web server. It is configured to listen on a non-standard TCP port, by default port number of 57772 (or another port that is not the usual, well-known, HTTP server port of 80). The private web server does not interfere with any other web server installation operating on the same host.

1.2.2 Configuring the Web Server and the CSP Gateway

The Caché installation performs web server and CSP Gateway configuration for common web servers and operating systems.

After installing Caché and the CSP Gateway, consult the CSP Gateway Configuration Guide to map file extensions for your system. This book also has configuration information for atypical CSP Gateway configurations.

To install the CSP Gateway on a remote server (that is, a system that is not running an instance of Caché), you can use one of two methods. On the remote server, you can run the

- Caché installation script and select to install Web Server only or
- Standalone CSPGateway installation script. The script asks for information about the remote Caché server: name, address, port, and optional password. The script automatically configures csp.ini based on this information.

After installing the CSP Gateway, consult the CSP Gateway Configuration Guide to map file extensions for your system.

Note: To prevent runtime errors, for High Availability configurations running over CSP, InterSystems recommends that you use a hardware load balancer with sticky session support enabled.

1.2.3 What You Should Know

To be productive with CSP, you should have some familiarity with the following:

- Caché objects and ObjectScript
- HTML
- JavaScript
- SQL

Some useful resources for learning HTML and JavaScript include:

- HTML v4.0.1 Specification

- HTML & XHTML: The Definitive Guide, published by O'Reilly.
- JavaScript: The Definitive Guide, published by O'Reilly.

1.2.4 CSP Samples

Caché comes with a set of sample CSP pages. To view these:

1. Start Caché.
2. To view the CSP samples, make sure that the web server on your machine is running.
3. Start your browser and go to the **CSP Samples Menu** (Use either the private web server at: http://localhost:57772/csp/samples/menu.csp or your external web server at: http://localhost/csp/samples/menu.csp.
4. If you installed Caché with normal or locked security features, a login page might be displayed. If so, log in.
5. Caché displays a list of sample CSP pages along with a short description of each. Click on any that interest you.

1.2.5 CSP Documentation

Documentation on CSP can be found here:

- *Using Caché Server Pages* describes how to create CSP pages
- *Using ZEN* describes how to use ZEN, a package that works on top of CSP for even more rapid web application development using prebuilt page objects. Zen documentation includes *Using Zen Components*, *Developing Zen Applications*, and *Using Zen Reports*.
- *CSP HTML Tag Reference*, a reference to all CSP tags
- *CSP Samples Menu*, shows many samples of CSP pages.
- *Caché Server Pages Quick Start Tutorial* gets you started.
- *CSP Web Applications Tutorial* is an in-depth tutorial.
- Class reference information for these classes:

  — %CSP.Page

  — %CSP.Session

To set up or configure CSP, see the following:

- *CSP Web Gateway Documentation*, online help on configuring the CSP Gateway, is available on the CSP Web Gateway Management page. In the Management Portal, navigate to **System Administration** > **Configuration** > **CSP Gateway Management** and click **Help**. By default, this takes you to the private web server. To see the CSP Web Gateway Management page for your production web server, substitute `localhost` or `localhost:`<*port_no*> for `localhost:57772` in the URL, for example:

  ```
  http://localhost/csp/bin/Systems/Module.cxw
  ```

- CSP Gateway Configuration Guide, when you install Caché, the CSP Gateway is installed automatically and works for most sites. If you need to configure the CSP Gateway manually, use the advanced configuration guide.

1.3 Creating Your First CSP Page

This section describes how to create a `Hello, World` CSP page in two different ways:

- Creating a class-based CSP page with web page objects.
- Creating an HTML-based CSP page using a marked-up HTML file.

1.3.1 Creating a Class-based CSP Page

Create a CSP page by creating a subclass of %CSP.Page and overriding its **OnPage** method. Any output written to the principal device by this method is automatically sent to a web browser and displayed as a web page.

To create a `Hello, World` CSP page programmatically, do the following:

1. Start Studio.
2. Select **File** > **New Project** to create a new project in the local database USER namespace.
3. Select **File** > **New** > **New Class Definition**.
4. On the first page of the Wizard, enter Test as the package name and Hello as the class name
5. On the second page, select `CSP` as the class type.
6. Click **Finish**. You see the new CSP class definition in the Studio Class Editor:

```
Class Test.Hello Extends %CSP.Page [ ProcedureBlock ]
{
    ClassMethod OnPage() As %Status
    {
        &html<<html>
        <head>
        </head>
        <body>>
        ;To do...
        &html<</body>
        </html>>
        Quit $$$OK
    }
}
```

7. In the **OnPage** method, replace the comment:

```
 ; To do...
```

   With a **Write** statement:

```
 Write "<b>Hello, World</b>",!
```

8. Save and compile the new class with **Build > Compile**.
9. Select **View > Web Page**

You see *Hello, World* displayed in the browser.

This CSP page, which is also a *CSP application* works as follows:

1. The browser sends a request for Test.Hello.cls to the local web server in the specified namespace.
2. The web server passes this request to the CSP Gateway which passes the request to a Caché CSP server. In our case, the browser, the web server, and the Caché Application server are all running on the same machine. In a real deployment, these would probably be on separate machines.
3. The CSP server looks for a class called Test.Hello and invokes its **OnPage** method.

4. Any output that the **OnPage** method writes to the principal device (using the **Write** command), is sent back to the browser (via the CSP Gateway and the web server).

These example shows the heart of CSP; the rest of CSP's functionality is built on top of this behavior.

The following is an example of adding more code. Insert the following lines after the line containing *Hello, World*:

```
Write "<ul>",!
For i = 1:1:10 {
   Write "<LI> This is item ", i,!
}
Write "</ul>",!
```

Now your page contains an unordered (bulletted) list of 10 items. Note that, in this context, Caché uses the exclamation point (!) character to write a carriage return to the principal device.

1.3.2 Creating an HTML Tag-based CSP Page

Another way to create a CSP page is to create an HTML file and let the CSP compiler transform it into a CSP class.

To create a `Hello.World` page using an HTML file, do the following:

1. Start Studio and select **File > New > CSP File > Caché Server Page**.
2. Replace the contents of the new CSP file with the following:

   ```
   <html>
   <body>
   <b>Hello, World!</b>
   </body>
   </html>
   ```

3. Select **File > Save**.
4. In the displayed **Save As** window, double-click csp/usr, the default CSP application.
5. Enter the filename Hello.csp and click **Save As** to save the file.
6. In the Studio window, select **View > Web Page**.

As with the previous example, you see **Hello, World!** displayed in the browser.

A CSP application can consist of a single CSP page or a set of pages. A CSP application acts as a unit, using settings that apply to the whole application. The system provides csp/user as the default CSP application. For more information on CSP applications, see the section "CSP Application Settings" in this book.

You can also create an HTML file using a text editor or HTML editor. Save this file as Hello.csp in the local directory *cachesys*/csp/user (where *cachesys* is where you installed Caché).

The Hello.csp page works as follows:

1. The browser sends a request for Hello.csp to the local web server
2. The web server passes this request to the CSP Gateway (connected to the web server) which, in turn, passes the request to a Caché CSP server.
3. The Caché CSP server looks for the file Hello.csp, and hands it to the CSP compiler.
4. The CSP compiler creates a new class called csp.Hello with an **OnPage** method that writes out the contents of the Hello.csp file. (It actually generates a set of methods each of which are, in turn, called from the **OnPage** method). This compilation step only occurs when the .csp file is newer than the generated class; subsequent requests are sent directly to the generated class.
5. The CSP server invokes the newly generated **OnPage** method and its output is sent to the browser as in the previous example.

As with the case of programmatic development, this is a purposefully oversimplified example included for pedagogical reasons. The CSP compiler is actually a specialized XML/HTML processing engine that can:

- Process server-side scripts and expressions in an HTML page
- Perform server-side actions when certain HTML tags are recognized.

As with the programmatic example, you can make this page more interesting by adding programming logic. For example:

```
<html>
<body>
<b>Hello, World!</b>
<script language="Cache" runat="server">
 // this code is executed on the server
 Write "<ul>",!
 For i = 1:1:10 {
    Write "<li> This is item ", i,!
 }
 Write "</ul>",!
</script>
</body>
</html>
```

As with the programmatic example, the resulting page displays an unordered (bulletted) list of 10 items.

2

CSP Architecture

This chapter covers the following topics:

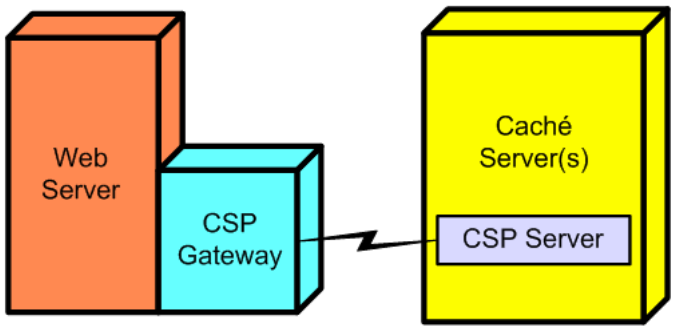
- The main "CSP components".
- "web server Configuration" for using CSP with supported web servers (see "Supported Web Servers" in the online *InterSystems Supported Platforms* document for this release).
- "CSP Gateway Configuration" to communicate with a Caché server.
- "CSP Application Options"

For more information on installing and configuring the CSP Gateway with your web server, reference the *CSP Gateway Configuration Guide*.

2.1 CSP Components: Web Server, CSP Gateway, CSP Server

CSP uses three software components: a web server, the CSP Gateway, and a CSP server (which runs on a Caché server):

Figure 2–1: CSP Components



The web server and the CSP server may be implemented by one or many computers. During development, all three components (web server, CSP Gateway, and CSP server) may be on a single PC. In a large scale deployment, there may be multiple web servers and CSP servers in two- or three-tier configurations.

This book treats these components (web server, CSP Gateway, CSP server) as though there were one of each. It describes CSP as though it were only serving HTML pages, although CSP can also serve XML pages, as well as other text formats and binary formats, such as images.

2.1.1 What Each Component Does

The web server is a software utility that does the following:

- Accepts incoming HTTP requests, usually from browsers
- Checks permissions
- Can serve static content
- Sends requests for CSP content (URLs ending in .csp or .cls) to the CSP Gateway

The CSP Gateway is a shared library, a .dll file, or a CGI script. It does the following:

- Determines which Caché server to send a request to.
- Sends requests to the correct Caché server.
- Maintains connections to the Caché server (to avoid having to continually open new connections).

The CSP server is a process running on a Caché server that is dedicated to serving requests from the CSP Gateway. It does the following:

- Receives an HTTP request for an application
- Checks the Application Configuration Settings (set in the Management Portal and saved in the .cpf file).
- Runs the associated class (either written directly or generated from a CSP page) which sends HTML back to the CSP Gateway which sends it to the web server and back to the browser.

2.1.2 Information Flow

CSP requests are processed by a standard web server (all the leading servers are supported) and the standard HTTP protocol. CSP manages communications between the web server and Caché and invokes application code to generate the page. The request and return process is as follows:

1. An HTTP client, typically a web browser, requests a page from a web server using HTTP.
2. The web server recognizes this as a CSP request and forwards it to the CSP Gateway using a fast server API.
3. The CSP Gateway determines the Caché server to talk to and forwards requests to the CSP server on that target system.
4. The CSP server running in Caché processes the request and returns a page to the CSP Gateway, which passes it back to the web server.
5. The web server sends it to the browser for display.

2.1.3 Static Files

The Caché database server serves all of CSP. The Caché database server can also serve any kind of static file for a web application via the CSP Gateway. In standard web applications, web servers (not database servers) typically serve static

content. To run Zen applications on Caché, you must configure your web server to allow the Caché server to serve all static files via the CSP Gateway.

Note: To run Zen-based applications, enable the **Serve Files** option and configure your web server to allow static files to be served by the Caché server. Then the Zen framework will be able to deliver dependent images and javascript helper documents.

Note: If you are serving files containing Unicode text, CSP uses the BOM to determine the correct encoding to use. The BOM must be present in Unicode text files.

Enabling the Serve Files Option

The **Serve Files** option for a CSP Application is either on (**Always**) or off (**No**) or the CSP Gateway can cache static files on the web server (**Always and cached**). The **Always and cached** setting improves efficiency as the system can serve a cached static page without going back to the Caché server). To run Zen applications, the **Serve Files** option must be enabled with either the **Always** or the **Always and cached** settings. (For more information, see the **Serve Files** option in the table "CSP Application Options".)

Configuring the Web Server to Allow Static Files to be Served by the Caché Server

Configure the web server to allow static files to be served by the Caché server.

By default, the Intersystems-provided installation script does not look for, or configure an external web server. If you run a Custom Install, you can choose the option to configure any previously-installed IIS or Apache web server to enable CSP support. The installation script creates the /csp virtual directory and creates mappings for the .csp, .cls, .zen and .cxw extensions to be handled by the gateway. This is sufficient for CSP to function normally via that web server. It does *not* enable support for the **Serve Files** feature. To make this happen, you must manually configure Apache or IIS to map specific file extensions to be handled via the Gateway. This is by design because opening the database server up to exposure via this mechanism is a security risk and should not be done invisibly.

See the section "Registering Additional File Types with CSP" in the CSP Gateway Configuration Guide for instructions for your web server.

Serving Static Files from the Web Server

You can still, if you choose to, use a traditional configuration of serving static pages from the web server, if, for example, you already have a web application set up this way. (In this case, the setting of the **Static Files** option is irrelevant.) This eliminates contention when a common web server serves two different versions of Caché, each requiring different versions of certain static files (for example, hyperevent broker components).

If you have configured the web server itself to serve static files, be sure that the static content is present on every single web server.

2.2 Web Server Configuration of a URL

Caché provides a default virtual directory of /csp from which to run your CSP applications. It also provides a default virtual directory of /cacheinstance/csp for use if you are running multiple instances of Caché. If you are running multiple instances of Caché and access a CSP application using the virtual directory/csp without the /cacheinstance before it, it accesses the last installed version of Caché. If you plan to allow all your CSP applications to be called with an application path that starts with either http://localhost/csp or http//localhost/cacheinstance/csp, then you do not need to make any changes in the web server configuration.

If want to create a CSP application with an application path that begins with something other than /csp or /cacheinstance/csp, then you need to make changes in the web server configuration file entry, `alias /csp`.

The following table shows the configuration files that need altering.

web server	Configuration Files
Apache, HP Secure web server	UNIX®: /etc/httpd/conf/httpd.conf Windows: <*web-server-install-directory*>\conf\httpd.conf
Sun	config/obj.conf and config/magnus.conf
Microsoft IIS	Define a virtual directory as described in "Configuring a New URL on a Microsoft IIS web server".

The following table shows examples of accessing a CSP page using combinations of web servers and Caché instances:

URL	web server	Instance
http://localhost/cache20071/csp/samples/menu.csp	public	cache20071
http://localhost/cache52/csp/samples/menu.csp	public	cache52
http://localhost/csp/samples/menu.csp	public	last version of Caché installed
http://localhost:57772/csp/sys/UtilHome.csp	private — used to access the Management Portal and online documentation	version of Caché installed using Webserver port of 57772

Note: CSP is case-sensitive. Specify your path names consistently when you are configuring CSP.

2.2.1 Configuring a New URL on a Microsoft IIS Web Server

Microsoft IIS is configured by defining a series of *virtual directories*. Each virtual directory consists of a name (which corresponds to the directory portion of a URL); a physical directory (the local directory in which static files, such as .html or .jpg, can be stored, if you opt to serve static files from the web server); and a set of permissions (such as **read** or **execute**).

Any request (URL) for CSP content includes a directory name. This directory name must correspond to either a virtual directory defined by the web server or a subdirectory of a virtual directory. This virtual directory must have at least **read** and **execute** privileges defined in order for CSP content to be served.

If you opt to serve static files from the web server, the web server looks for static files (such as .html or .jpg) in the *physical* directory defined for the virtual directory. Neither the web server nor the Caché server looks for .csp files in the physical directory; .csp files are stored in the machine on which the Caché server is running. If both the web server and Caché are running on the same machine (as is recommended during application development, for example) then both may, coincidentally, look in the same location for static and .csp files — and this is how Caché configures itself and the local web server during installation.

During installation, Caché detects if an IIS server is running and attempts to configure it to define a virtual directory named /csp. This is how requests to both /csp/samples and /csp/user (which are subdirectories of /csp) are both sent to the local Caché installation.

If you add a new CSP application, you do not have to perform any IIS configuration if the URL path for the new application also starts with /csp. For example, /csp/myapp uses the IIS virtual directory defined for /csp. If you do not want your application path to start with /csp, then you need to define a new virtual directory for IIS that corresponds to your application path.

For example, to define a CSP application that uses the URL path /myapp, do the following:

1. Open the IIS manager (note: every version of Windows has a unique way to do this; typically this is available via the Windows Control Panel).
2. Define a virtual directory called /myapp by right-clicking **Default Web Site** and selecting **New > Virtual Directory**.

   (For an example of how to do this, see the "Add Virtual Directories in IIS" section in the *CSP Gateway Configuration Guide*).
3. Grant `read` and `execute` permissions for this directory.
4. If you want the web server to serve static content, specify the physical directory in which you plan to store static content.

   You also have to perform additional CSP Gateway and CSP server configuration as described in the following sections.

2.3 CSP Gateway Configuration

The CSP Gateway is a DLL or shared library installed on and loaded by the web server. The CSP Gateway detects any requests for files with a .csp or .cls extension and sends them to a defined Caché server for processing.

2.3.1 CSP Gateway Manager

You can configure the CSP Gateway using the CSP Gateway Manager (the CSP Web Gateway Management page) or by editing its configuration file, csp.ini, directly.

The CSP Gateway Manager is a small web application that you can use from your browser. You can access the CSP Gateway Manager by navigating to **System Administration** > **Configuration** > **CSP Gateway Management**. By default this accesses the CSP Gateway Manager for the private web server.

To access the CSP Gateway Manager for your production web server, change the URL by substituting `localhost` or `localhost:`<*port_no*> as necessary.

For information on configuring the CSP Gateway, navigate to **System Administration** > **Configuration** > **CSP Gateway Management** and click **Help**. For more detailed information, see *CSP Gateway Configuration Guide*.

Note: Localization of the CSP Web Gateway Management pages is based solely on the contents of the CSPres.xml installed (if any). If no localization file is present then the CSP Web Gateway Management pages default to using the embedded English text. The language settings of the browser have no influence on this mechanism.

2.3.2 Defining Server Access

Note: To prevent runtime errors, for High Availability configurations running over CSP, InterSystems recommends that you use a hardware load balancer with sticky session support enabled.

You can define a list of servers (Caché or Ensemble servers that might run CSP applications) that this CSP Gateway can access. Each server has a logical name, a TCP/IP address, a TCP/IP port number (the default is 1972), and an enabled or disabled flag. In addition, you can configure the minimum and maximum number of connections made to this server as well as timeout and logging values.

Since each server has a logical name, the CSP Gateway makes it easy to connect applications to particular servers by name and later change the characteristics of a server in one place without having to reconfigure every application using the server.

After the initial installation, the CSP Gateway has one logical server defined, `LOCAL`, which is defined to connect to the local copy of Caché.

To add one or more servers that you want the CSP Gateway to be able to access, open the CSP Gateway Manager as described in the previous section and click **Server Access**. See the section "Accessing CSP on Multiple Caché Servers" in the *CSP Gateway Configuration Guide* for details of the procedure.

Example of the default LOCAL server in the CSP.ini file:

```
LOCAL=Enabled
...
[LOCAL]
Ip_Address=127.0.0.1
TCP_Port=1972
Minimum_Server_Connections=3
```

2.3.3 Defining Application Access

Note: A Caché installation creates a new /csp configuration. If you have configured /csp as your application, your configuration is overwritten when you install a new build of Caché. To maintain your application configuration, enter a path other than /csp.

Any directory under /csp works fine, such as /csp/myapplication, but the path cannot contain any dots (periods). These lead to ambiguity for the CSP Gateway. In this example: /csp/samples/menu.csp/csp/aaa/bbb/ccc.cls, the CSP Gateway could either interpret this as a request for /csp/samples/menu.csp/csp/aaa/bbb/ccc.cls or as a REST request for/csp/samples/menu.csp (where PATH_INFO is /csp/aaa/bbb/ccc.cls). The Gateway, working in the web server environment, has no way of resolving these ambiguities.

CSP is case-sensitive. Specify your path names consistently when you are configuring CSP.

A *CSP application* is a set of pages or classes that are addressed using a given URL. For example, all the CSP sample pages are part of the /csp/samples application. An application may contain subdirectories, for example /csp/samples/cinema.

The CSP Gateway Manager lets you define the URL path that a CSP application uses to connect to a Caché server. CSP considers all files within a particular URL directory (or subdirectory thereof) to be part of the same application.

By default, the CSP Gateway defines a single application path, /csp, which sends all CSP requests to the logical server LOCAL. Requests to /csp/samples and /csp/user are both sent to the local Caché installation.

If you create a new CSP application with a URL that starts with /csp, you do not have to change the CSP Gateway configuration. Your new application, such as /csp/myapp, uses the CSP application settings defined for /csp. If you do not want your URL path to start with /csp, then you need to define a new CSP application within the CSP Gateway that corresponds to your URL path.

For example, to define a CSP application that begins with the URL path /myapp, do the following:

1. Open the CSP Gateway Manager by navigating to **System Administration** > **Configuration** > **CSP Gateway Management** in the Management Portal.
2. Select **Application Access**.
3. Click **Add Application**.
4. Enter /myapp into the **Application Path** field.
5. Select the **Default Server** where the application is from the list (these are defined in the "Defining Server Access" section).
6. Click **Submit** to save the /myapp application access configuration.

For details on the other fields on the **Application Access** page, click **Help**.

2.3.4 CSP Gateway Parameters

The CSP Gateway has a number of parameters that you can adjust. These include timeout values, failover and load-balancing characteristics, and CGI environment variables.

For details on these parameters, see the "Operation and Configuration" chapter in the *CSP Gateway Configuration Guide* or click **Help** on the CSP Web Gateway Management page.

2.4 CSP Application Settings

When a CSP server receives an incoming HTTP request, it uses the local Caché CSP application settings to determine how to process the request. This section describes how Caché processes CSP application requests using CSP application settings.

2.4.1 Enabling Application Access to %CSP Pages

By default, a user application can access:

- All non-% classes
- Pages of the /csp/sys/ application and all of its subapplications.
- Pages of the isc/studio/templates/ and /isc/studio/usertemplates/ applications.

A user application can also access the following classes:

- %CSP.Broker, %CSP.StreamServer, %CSP.PasswordChange, %CSP.PageLookup, and %CSP.Login

  Important: If you are creating a custom login page, it must be a subclass of %CSP.Login.

- %ZEN.SVGComponent.svgPage, %ZEN.Dialog.* are allowed.
  - All other %ZEN.* classes are *not* allowed.
  - All other %Z* classes *are* allowed.
- All %z* classes are allowed.

Checking for allowed classes is performed in addition to checking the setting in the CSP application. You can view and change the application settings by navigating to **System Administration** > **Security** > **Applications** > **Web Applications** on the Management Portal. So a class reference must pass both sets of tests in order to be allowed. (See in the **Permitted Classes** setting in the table in the section "Editing Web Application Settings" in this book.)

To permit access to additional classes, configure the global `^SYS("Security","CSP","category")` *in the %SYS namespace*, where *category* is `AllowClass`, `AllowPrefix`, or `AllowPercent`. The following sections describe these options.

Important: Checking is done by applying the default rules first, then the categories in the order listed.

Also, each keyword can be invoked more than once. This means that you can make an entire package accessible, and then restrict access to one class in that package.

2.4.1.1 Background Information on the ^SYS Global

The ^SYS global is available in the %SYS namespace and contains configuration information, including the details described in this book. You may find it helpful to start by examining the current contents of the relevant part of this global. To do so, open the Terminal and switch to the %SYS namespace. Then enter the following command:

```
zw ^SYS("Security", "CSP")
```

The system then displays one line for each node, showing its current value. For example:

```
^SYS("Security","CSP")=1
^SYS("Security","CSP","AllowClass","/csp/samples/","%CSP.UI.Portal.About")=1
^SYS("Security","CSP","AllowClass","/csp/samples/","%SOAP.WebServiceInfo")=1
^SYS("Security","CSP","AllowClass","/csp/samples/","%SOAP.WebServiceInvoke")=1
^SYS("Security","CSP","AllowPrefix","/csp/samples/","%DeepSee.")=1
```

2.4.1.2 Category: AllowClass

If your application relies on invoking a particular class, use the `AllowClass` option to make that class available.

Important: If your application relies on invoking any class other than those listed above, it could potentially be unsafe to use. InterSystems recommends that you determine if calling this class is required, and perform a risk assessment for your deployment, so that you understand the implications of making the class available.

To enable a given web application to invoke a particular class, use the following command in the %SYS namespace:

```
Set ^SYS("Security", "CSP", "AllowClass", "web-app-name", "package.class") = value
```

Where:

- *web-app-name* is the name of the web application, *followed by a trailing slash*.

  To enable *all* web applications to use the given class or package, specify *web-app-name* as 0; in this case, you can omit the enclosing quotes.

- *package.class* is the fully qualified name of a class. If you omit *class*, then *all* classes in the specified package are allowed.

- *value* is either 1 or 0.

  If you specify this as 1, the web application can invoke this class (or package).

  If you specify this as 0, this web application cannot invoke this class (or package).

For example, to enable the /csp/webapps application to use the class %User.Page, you would use the following command:

```
Set ^SYS("Security", "CSP", "AllowClass", "/csp/webapps/", "%User.Page") = 1
```

Or to enable *all* web applications to use the %User.Page, you would use the following command:

```
Set ^SYS("Security", "CSP", "AllowClass", 0, "%User.Page") = 1
```

For another example, to enable the /csp/myapp application to use all classes in the %User package except for the %User.Other class, you would use the following two commands:

```
Set ^SYS("Security", "CSP", "AllowClass", "/csp/myapp/", "%User") = 1
Set ^SYS("Security", "CSP", "AllowClass", "/csp/myapp/", "%User.Other") = 0
```

2.4.1.3 Category: AllowPrefix

If your application relies on invoking multiple classes or packages that begin with the same set of characters, use the `AllowPrefix` option.

Important: If your application relies on invoking any class other than those listed above, it could potentially be unsafe to use. InterSystems recommends that you determine if calling this class is required, and perform a risk assessment for your deployment, so that you understand the implications of making the class available.

To enable a given web application to invoke classes or packages that begin with the same set of characters, use the following command in the %SYS namespace:

```
Set ^SYS("Security", "CSP", "AllowPrefix", "web-app-name", "prefix") = value
```

Where:

- *web-app-name* is the name of the web application, *followed by a trailing slash*.

  To enable *all* web applications to use the given classes or packages, specify *web-app-name* as 0; in this case, you can omit the enclosing quotes.

- *prefix* is the first characters in the name.

- *value* is either 1 or 0.

  If you specify this as 1, the web application can invoke these classes (or packages).

  If you specify this as 0, this web application cannot invoke these classes (or packages).

For example, to enable the /csp/webapps application to invoke the entire MyApp package, use the following command:

```
Set ^SYS("Security", "CSP", "AllowPrefix", "/csp/webapps/", "MyApp.") = 1
```

Note that *prefix* is `"MyApp."` and the period in the prefix means that the web application cannot access the packages such as `MyAppUtils`. The web application can, however, access the packages `MyApp.Utils` and `MyApp.UnitTests`.

For another example, to enable *all* applications to access *all* packages that begin with `My`, use the following command:

```
Set ^SYS("Security", "CSP", "AllowPrefix", 0, "My") = 1
```

For another example, suppose that the /csp/myapp application should be able to access all classes in the %MyPkg package except for the class %MyPkg.Class1. In that case you would use the following two commands:

```
Set ^SYS("Security", "CSP", "AllowClass", "/csp/myapp/", "%MyPkg.Class1") = 0
Set ^SYS("Security", "CSP", "AllowPrefix", "/csp/myapp/", "%MyPkg.") = 1
```

2.4.1.4 Category: AllowPercent

If your application relies on invoking the packages that begin with the % character generally, the `AllowPercent` option makes those classes available.

Important: If your application relies on invoking any class other than those listed above, it could potentially be unsafe to use. InterSystems recommends that you determine if calling this class is required, and perform a risk assessment for your deployment, so that you understand the implications of making the class available.

To enable all web applications to use all packages that begin with the % character, use the following command in the %SYS namespace:

```
Set ^SYS("Security", "CSP", "AllowPercent") = 1
```

Note: Or use the value 0 to explicitly forbid any web application from accessing these packages.

2.4.1.5 Special Case: DeepSee

For a web application to use DeepSee, it needs access to all the classes in the %DeepSee package. To enable a particular application to use DeepSee, use the following command in the %SYS namespace:

```
Do EnableDeepSee^%SYS.cspServer("/csp/webapp/")
```

Where *web-app-name* is the web application's name with a trailing slash. The previous command is equivalent to the following commands:

```
Set ^SYS("Security","CSP","AllowClass","web-app-name","%DeepSee.") = 1
Set ^SYS("Security","CSP","AllowClass","web-app-name","%CSP.UI.Portal.About")=1
```

Where *web-app-name* is the web application's name with a trailing clash. Note that the first line uses `%DeepSee.` with a trailing period.

Or to enable all applications to use DeepSee, use the following variation:

```
Do EnableDeepSee^%SYS.cspServer(0)
```

For example, to enable the /csp/webapp web application to use DeepSee, use the following command:

```
Do EnableDeepSee^%SYS.cspServer("/csp/webapp/")
```

To disallow a specific web application from using DeepSee, use the following command:

```
Set ^SYS("Security", "CSP", "AllowPrefix", "web-app-name", "%DeepSee.") = 0
```

2.4.2 Editing Web Application Settings

You can create or modify settings for how you want Caché to process a specific CSP application on the **Edit Web Application** page of the Management Portal as follows:

1. Select **System** > **Security** > **Web Applications**.

   This lists configured web applications. The **Type** column identifies an application as a user application (`CSP`) or a system application (`CSP,System`; a CSP-based utility included with Caché).

2. Select an application, click **Edit**, and enter or change the information.

3. When finished with edits, restart Caché for the new settings to take effect.

The **General** tab holds fields that specify information needed for basic operation of the application. See "Editing an Application" in the *Caché Security Administration Guide* for more information on these fields.

The **Application Roles** tab lets you select roles to which to assign the user during use of the application. The **Application Roles** that you select here are added to the set of roles to which the user is already assigned.

The **Matching Roles** tab lets you assign the application user to additional roles during use of the application, based on current role assignments.

For more on roles and custom login pages, see "Applications" in the book *Caché Security Administration Guide*.

2.4.3 Defining a New Application

To define a new CSP application named /myapp on a CSP server, follow the steps below:

1. In the Management Portal, select **System** > **Security** > **Web Applications** and click **Create New web application**.

2. Type in the URL for the new application name, /myapp in this case, and click **OK**.

3. Fill in any needed application properties (most are optional). (See the table in the section Application Settings.)The most important are:

   - **Enable/Disable Authentication allowed** — the valid authentication technologies for connecting to the application
   - **Namespace** — the Caché Namespace in which this application is run

- **Caché Physical Path** — the physical location of CSP files (if you are using HTML-based development)

4. Click **Save**.
5. Click the **Application Roles** tab to select roles to assign the user to during use of the application. These **Application Roles** are added to the set of roles the user is already assigned to.
6. Click the **Matching Roles** tab to assign the application user to additional roles during use of the application, based on current role assignments.

3

HTTP Requests in CSP

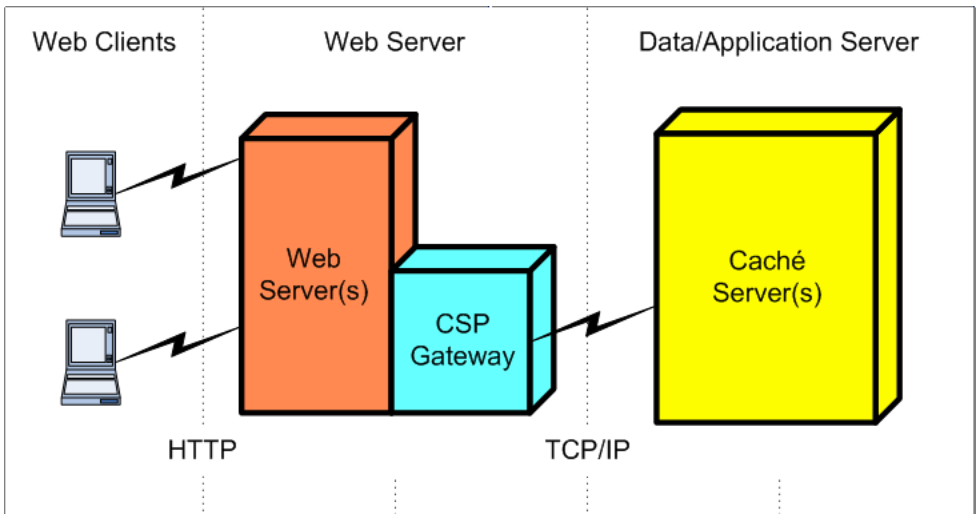
The primary task of CSP is to provide dynamic content in response to incoming HTTP (Hypertext Transport Protocol) requests. This section covers the basics of how HTTP requests are handled by CSP.

HTTP is a simple protocol in which a client makes a request to a server. HTTP is a stateless protocol; the connection between a client and a server lasts only as long as it takes to service the request. Every HTTP request contains a *request header* which specifies the request type (such as GET or POST), a URL, and a version number. A request may also contain additional information. CSP automatically determines which HTTP requests it should handle, dispatches them to the appropriate class running on a Caché server, and packages up the request information into easy-to-use objects (such as the %CSP.Request object).

3.1 CSP Runtime Environment

The following diagrams show the architecture of CSP and HTTP requests:

Figure 3–1: CSP Architecture



The runtime environment of a CSP application consists of the following:

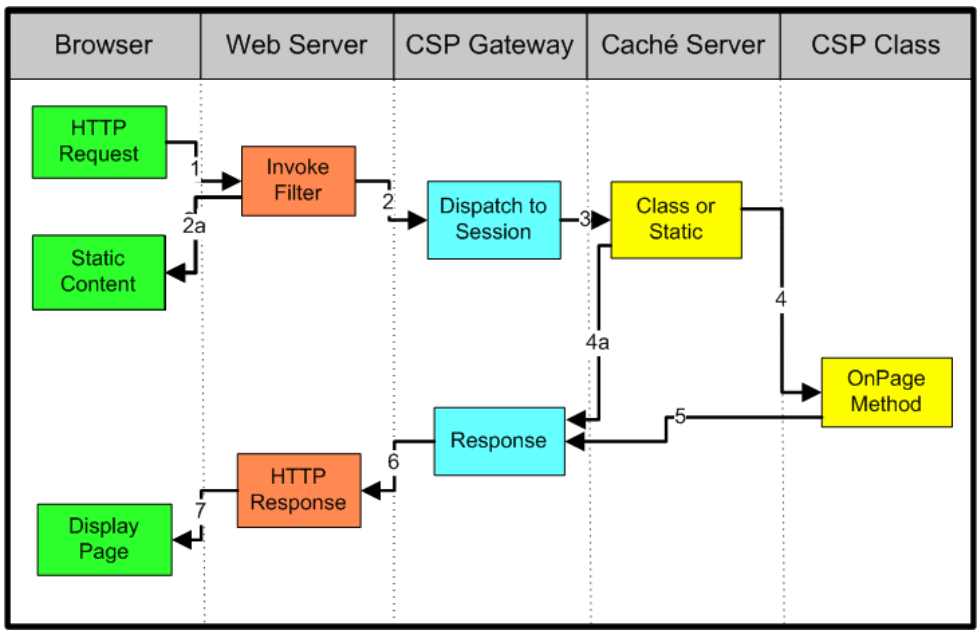
- An HTTP Client (such as a web browser)
- An HTTP Server (a web server such as Apache or IIS)
- The CSP Gateway (a Caché add-on to the web server)

- The Caché server (on which the CSP server runs the requested CSP application)

3.2 HTTP Request Processing

The following diagram illustrates the flow of events when CSP processes an HTTP request:

Figure 3–2: HTTP Event Flow



1. The browser (or similar web client) makes an HTTP request.
2. The web server determines that this is a CSP request and dispatches it to the CSP Gateway (installed on the web server). 2a The web server might serve static content, depending on your application configuration.
3. The CSP Gateway repackages the request and sends it to the correct Caché Server.
4. The Caché Server decodes the message and determines whether the request is for a static page or a CSP class.

   If the request is for a static file (such as an .html or .jpg file), the Caché server finds the file within the local file system and sends its contents back to the client. (Note that if you are serving files containing Unicode text, CSP uses the BOM to determine the correct encoding to use. The BOM must be present in Unicode text files.)

   If it is for a class, it determines which event handling class (part of your application) will process the event and invokes the class **Page** method.
5. The output of the **Page** method or the static page is sent back to the CSP Gateway as an HTTP response.
6. The CSP Gateway hands the HTTP response to the web server (specifically, the response is streamed back to the web server via the CSP Gateway).
7. The web server returns the response to the web browser which then processes the response — in the case of HTML, it displays it.

3.2.1 Web Server and the CSP Gateway

An HTTP request starts when an HTTP client, such as a web browser, sends a message to a web server. The CSP Gateway is a DLL or shared library used by the web server (such as IIS or Apache) to process certain types of events. The CSP Gateway processes an HTTP request if the following conditions is true:

- The directory path of the URL has the correct access privileges defined within the web server.

The CSP Gateway provides the following functionality:

1. It provides minimal processing and off-loads most of its work to the Caché server providing more resources for the web server.
2. It maintains a pool of connections to named CSP servers.
3. It provides failover options to allow use of multiple, interconnected CSP servers.

3.2.2 CSP Server

The CSP server is a process running on a Caché server that is dedicated to serving requests from a CSP Gateway. Each Caché server may run as many CSP server processes as desired (subject to limits imposed by machine type; CSP servers are not counted in license calculations).

When handling *stateless* requests, each CSP server process can support requests from many different clients. In `state preserving` mode, a process is dedicated to process requests from one client, until `state preserving` mode is turned off.

Note: One of the key strengths of Caché is that there is no real difference between an *application server* and a *data server*; you can configure your application to use as many, or as few, machines as necessary based on your requirements. This is done independently of application logic and database schema. Whether a particular system is an application server or a data server (or both) is simply a matter of configuration.

3.2.3 CSP Server Event Flow

When the CSP server receives a request from the CSP Gateway, it determines whether the request is for a static page or for a CSP class. If it is for a static page, it sends the page back immediately. If it is for a CSP class, it does the following:

1. Determines what session this request belongs to. If none, it starts a new session.
2. Makes sure that the request is processed in the correct Caché Namespace.
3. Makes sure that the correct %CSP.Session object is available and creates an instance of the %CSP.Request object based on the information contained in the HTTP request. If any decryption is required, it does that as well.
4. Constructs %CSP.Response object to allow application to modify response headers.
5. Determines which class should handle the request and invokes its **Page** method (which, in turn, invokes the **OnPage** callback method).

3.2.4 CSP Server URL and Class Name Resolution

The CSP server determines which class to dispatch an HTTP request to by interpreting its URL. CSP disassembles a URL into the following components:

URL: `http://localhost:[<port_no>]/csp/samples/object.csp?OBJID=2`

Table 3–1: URL Components

Component	Purpose
`http://`	protocol
`localhost`	web server address
[<*port_no*>]	optionally, the port number that the web server is running on; defaults to port 80
/csp/samples/	directory
object.csp	file name and extension
`?OBJID=2`	query

The protocol and server address are handled by the web server and are not relevant to the CSP server. The directory is used to determine which CSP application the URL refers to. Every Caché configuration can define a number of CSP applications, which are identified by the directory portion of a URL. Each CSP application specifies a number of settings used by all requests with a given URL directory. The most important of these are the Caché Namespace in which the request is to be executed and the authentication mechanism, which specifies what kind of connections to the application can be established. To create and modify CSP applications, navigate to **System Administration** > **Security** > **Applications** > **Web Applications** on the Management Portal.

The name of the class to handle the request is determined from the file name using the following algorithm, based on the CSP application:

- If the file extension is .cls, then use the file name as a class name.
- If the file extension is .csp, then create a class name using `csp` (or the package name specified in the configuration) as a package name and the file name as the class name. If this class does not exist, or is out of date, then the CSP compiler creates a class from a CSP source file (if `autocompile` is on). This file has the same name and extension as that in the URL.

For example, the CSP server dispatches this URL:

```
http://localhost:57772/csp/samples/menu.csp
```

to a class called menu contained within the package `csp` running in the Caché namespace associated with the directory /csp/samples (in this case the `SAMPLES` namespace).

If the URL directory /csp/accounting is associated with the Caché namespace `ACCOUNTING`, then the CSP server dispatches this URL:

```
http://localhost:57772/csp/accounting/Ledger.csp
```

to a class called ledger contained within the package `csp` running in the Caché namespace `ACCOUNTING`.

Note that URL file names used with CSP have a number of restrictions:

- They must be valid Caché class names (they cannot contain white space or punctuation characters (except for dots and percent characters (`%25`)) and cannot start with numeric characters).
- They must not conflict with other class names already in use.

Note: If a .csp file is placed within a subdirectory of a defined directory, then the subdirectory name becomes part of the package name for the %CSP.Page class used for the page. For example, if the URL directory /csp/samples is defined as a CSP application, then `/csp/samples/myapp/page.csp` refers to a class named csp.myapp.page.

3.3 %CSP.Page Class

On the CSP server, all HTTP requests are handled by invoking methods defined by the %CSP.Page class. The %CSP.Page class never directly handles requests itself; it simply defines the interface required to process HTTP requests. The actual event handling is done by a subclass of %CSP.Page (created either manually or as a result of processing a CSP source file).

Subclasses of %CSP.Page are never instantiated; that is, no %CSP.Page objects are ever created. The methods defined by %CSP.Page are all class methods and do not require an object in order to be invoked. As we shall see, any state information required by these methods is provided by other objects (such as the %CSP.Request and %CSP.Session objects) which are managed by the CSP server.

3.3.1 Page Method

After the CSP server determines which %CSP.Page class should process a request, it sets up the appropriate processing context and then invokes that class' **Page** method. Setting up the processing context includes redirecting the standard output device (*$IO*) so that all output (using the **Write** command) is sent back to the HTTP client and creating instances of any required objects (such as the *%request*, *%response*, and *%session* objects) or local variables.

The **Page** method handles the complete response to the HTTP request. It does this by invoking the callback methods, **OnPreHTTP**, **OnPage**, and **OnPostHTTP** in that order. These are referred to as callback methods because a subclass can override them in order to provide custom behavior.

The **OnPreHTTP** method is responsible for writing out the header for the HTTP response. This includes information such as content type and cookies. The default behavior is to set content type to `text/html`. You typically only need to override the **OnPreHTTP** method in cases where you need more direct control over the response header.

The **OnPage** method performs the bulk of the effort in responding to an HTTP request. It is responsible for writing out the body of the request, such as an HTML or XML document.

For example, here is a sample CSP class containing an **OnPage** method:

```
Class MyApp.Page Extends %CSP.Page
{
ClassMethod OnPage() As %Status
{
    Write "<html>",!
    Write "<body>",!
    Write "My page",!
    Write "</body>",!
    Write "</html>",!
    Quit $$$OK
}
}
```

The **OnPostHTTP** method is provided as a place to perform any operations you wish to perform after processing of the HTTP request is complete.

3.3.2 %CSP.Page Class Parameters

The %CSP.Page class contains a number of class parameters that you can override to provide custom behavior without having to write code.

For a list of all available class parameters, refer to the documentation for %CSP.Page.

If you are developing applications programmatically you can override these class parameters within the subclasses of %CSP.Page you create (using the Class Editor within the Studio for example).

If you are creating pages using .csp files, you can provide values for these parameters using the `csp:class` tag:

```
<csp:class PRIVATE="1">
```

Limiting Page Access by Resources

Use the **SECURITYRESOURCE** parameter to limit access to CSP pages. **SECURITYRESOURCE** takes a comma-delimited list of system resources and associated permissions. You can specify an OR condition using the vertical bar (|) and an AND condition using a comma (,). A user must hold the specified permissions on all of the specified resources in order to view this page or invoke any of the page's server-side methods from the client.

An item in the list has the following format:

```
Resource[:Permission]
```

Resource is any of the resources set on this system. Navigate to **System Administration** > **Security > Resources** for a list of resources.

Permission is one of USE, READ, or WRITE. Optional; default is USE.

Example

```
R1,R2|R3,R3|R4
```

This example means the user must have resource R1 AND one of (R2 OR R3) AND one of (R3 OR R4). If the user has R1,R3 they can run the page. If the user has R1,R4, they cannot run the page, as they do not meet the R2 OR R3 condition. The vertical bar (|) OR condition takes precedence over the comma (,) AND condition.

3.3.3 Handling CSP Errors

%CSP.Error is the default CSP error page. Use this as the superclass of any error pages that you create. You can pull information out of the error using the functions provided in %CSP.Error.

3.3.3.1 Handling CSP Errors Before a License Has Been Granted

If you already have an existing session and the user tries to go to a page that is not found, CSP displays the standard error page because the session already has a license.

If a CSP application does not yet have a license, and any of the following errors occur, then CSP displays the standard web `HTTP/1.1 404 Page Not Found` error message by default. You can change what page is displayed when errors are encountered before a license has been granted by setting the following parameters on the error page (usually a subclass of %CSP.Error) for your application.

LICENSEERRORPAGE

If the following error is generated, CSP looks at the value of the **LICENSEERRORPAGE** parameter:

```
Cannot grant license.
```

LICENSEERRORPAGE can have the following two values:

"" — Returns the `HTTP/1.1 404 Page Not Found` error (default)

Path to a static HTML page — Displays the named static page, such as /csp/samples/static.html.

PAGENOTFOUNDERRORPAGE

If any of the following errors are generated, CSP looks at the value of the **PAGENOTFOUNDERRORPAGE** parameter:

```
Class does not exist
Method does not exist
CSP application does not exist (set parameter on default error page)
CSP page does not exist
File does not exist
CSP namespace does not exist
CSP illegal request
File cannot open
CSP session timeout
```

PAGENOTFOUNDERRORPAGE can have the following three values:

"" — Return the HTTP/1.1 404 Page not found error (default)

1 — Obtains a license and displays the standard error page.

Path to a static HTML page — Displays the named static page, such as /csp/samples/static.html.

OTHERSTATICERRORPAGE

If any other errors are generated, CSP looks at the value of the **OTHERSTATICERRORPAGE** parameter.

OTHERSTATICERRORPAGE can have the following three values:

"" — Obtains a license and displays the standard error page (the default)

1 — Outputs the 404 Page not found error, no license required.

Path to a static HTML page — Displays the named static page, such as /csp/samples/static.html.

3.4 %CSP.Request Object

When the CSP server responds to an HTTP request, it packages information about the incoming request into an instance of the %CSP.Request object. You can refer to this object using the variable *%request*. Refer to the documentation for the %CSP.Request class for a complete list of its properties and methods.

3.4.1 URL Property

To find the URL (not including the query string) of an incoming HTTP request, use the URL property of the %CSP.Request object:

```
Write "URL: ", %request.URL
```

3.4.2 Data Property and URL Parameters

A URL may contain a list of parameters (also known as the URL query). The %CSP.Request object makes these available via its Data property.

For example, suppose the incoming URL contains:

```
/csp/user/MyPage.csp?A=10&a=20&B=30&B=40
```

You can retrieve these parameters on the server using:

```
Write %request.Data("A",1)    // this is 10
Write %request.Data("a",1)    // this is 20
Write %request.Data("B",1)    // this is 30
Write %request.Data("B",2)    // this is 40
```

Data is a multidimensional property and each value stored within it has 2 subscripts: the name of the parameter and the index number of the parameter (parameters can occur multiple times within a URL as with B above). Note that parameter names are case-sensitive.

Also note that it does not matter if an incoming HTTP request is a GET or a POST request: the Data property represents the parameter values in exactly the same way.

You can use the ObjectScript **$Data** (**$D**) function to test if a given parameter value is defined:

```
If ($Data(%request.Data("parm",1))) {
}
```

If you wish to refer to a parameter but are not sure if it is defined, you can use the ObjectScript **$Get** function:

```
Write $Get(%request.Data("parm",1))
```

You can find out how many values are defined for a particular parameter name using the **Count** method of the %CSP.Request object:

```
For i = 1:1:%request.Count("parm") {
   Write %request.Data("parm",i)
}
```

3.4.3 CgiEnvs Property and CGI Environment Variables

The web server provides a set of values, referred to as CGI (Common Gateway Interface) environment variables, which contain information about the HTTP client and web server. You can get access to these CGI environment values using the multidimensional property CgiEnvs. You can use this in the same manner as the Data property.

For example, to determine what type of browser is making the HTTP request, look at the value of the CGI environment variable *HTTP_USER_AGENT*:

```
Write %request.CgiEnvs("HTTP_USER_AGENT")
```

For information on the CGI environment variables you can use, see the section "CGI Environment Variables" in the *CSP Gateway Configuration Guide*.

3.4.4 Cookies Property

If the HTTP request contains any cookies, you can retrieve their values using the multidimensional property, Cookies. You can use this in the same manner as the Data property.

In this book, see section "%CSP.Response Object and the OnPreHTTP Method" for examples of setting cookies.

Data can also be saved in the *%session* object. See "User Session Data – Data Property". See an example of using cookies in cookie.csp in the CSP Samples database.

3.4.5 MIME Data Property

An incoming request may contain MIME (Multipurpose Internet Mail Extensions) data. This is typically used for larger pieces of information, such as files. You can retrieve MIME data using the %CSP.Request object. This creates and returns an instance of a Caché stream object that you can then use to read the MIME data.

For an example using MIME data, refer to the upload.csp page in the CSP samples.

3.5 %CSP.Response Object and the OnPreHTTP Method

You can control what response headers are sent back to the HTTP client using the %CSP.Response object. The CSP server automatically creates an instance of this class and places a reference to it in the variable *%response*.

As the *%response* object controls HTTP headers, you typically set its properties in the **OnPreHTTP** method of the %CSP.Page class. For example, to redirect an incoming HTTP request, define the following **OnPreHTTP** method:

```
Class MyApp.Page Extends %CSP.Page
{
// ...

ClassMethod OnPreHTTP() As %Boolean
{
    Set %response.ServerSideRedirect = "C:\CacheSys\csp\samples\redirect.csp"
    Quit 1
}
}
```

If you use CSP classes to create a CSP page, use the following code to set header values:

```
Class MyPage Extends %CSP.Page
{
ClassMethod OnPreHTTP() As %Boolean
{
    Do %response.SetCookie("name","value")
    Quit 1
}
```

If you use HTML to create a CSP page, use the following code to set header values:

```
<head></head>
<script language="Cache" method="OnPreHTTP" arguments="" returntype="%Boolean">
    Do %response.SetCookie("name","value")
    Quit 1
</script>
<body></body>
```

3.5.1 Serving Cookies with the SetCookie Method

You can send cookies to the HTTP client using the *%response* object's **SetCookie** method. Refer to the section "Storing Data in Cookies".

3.5.1.1 Accessing Cookies

Once a cookie has been saved. you can access its value with this code:

[in Page Classes]

```
ClassMethod OnPage() As %Status
{
Write "<body>"
Write "<p>COOKIES:</p>"
Write "<ul>"
Set cookie=%request.NextCookie("")
While cookie'="" {
      For count=1:1:%request.CountCookie(cookie) {
   Write "<li>",cookie," - ",..EscapeHTML(%request.GetCookie(cookie,count)),"</li>",!
   }
Set cookie=%request.NextCookie(cookie)
}
Write "</ul>"
Write !,"</body>"
Quit $$$OK
}
```

[on CSP pages]

```
<body>
<p>COOKIES:</p>
<ul>
<script language="Cache" runat="server">
Set cookie=%request.NextCookie("")
While cookie'="" {
    For count=1:1:%request.CountCookie(cookie) {
    Write "<li>",cookie," - ",..EscapeHTML(%request.GetCookie(cookie,count)),"</li>",!
    }
Set cookie=%request.NextCookie(cookie)
}
</script>
</ul>
</body>
```

A cookie definition can include an expiration date and a path in this format:

```
Do %response.SetCookie("NAME","VALUE",expireData,path)
```

A blank *expireData* field defines an in-memory cookie. If, however, you specify a value for the *expireData* field this becomes a permanent cookie that is removed at the time specified. The format for the *expireData* field is `Wdy, DD-Mon-YYYY HH:MM:SS GMT`, for example: `Wednesday, 24-Mar-2004 18:12:00 GMT`.

3.5.2 Serving Different Content Types

Typically a CSP page serves `text/html` content. You can specify a different content type in several ways:

- By setting the value of the %CSP.Page class parameter **`CONTENTTYPE`** within your page class.
- By setting the value of the *%response* object's ContentType property within your page's **OnPreHTTP** method.

4

CSP Session Management

HTTP is a stateless protocol; every request has no knowledge of previous requests. While this works well for web sites that provide users with simple static content, it makes it difficult to develop interactive, dynamic web applications. To help with this, CSP provides what is called *session management.*

4.1 Sessions with CSP.Session

A *session* represents a series of requests from a particular client to a particular application over a certain period of time.

CSP provides session tracking automatically; you do not have to do anything special to enable it. CSP applications can inquire and modify aspects of their session by means of the %CSP.Session object. The CSP server makes this object available via the ObjectScript *%session* variable. For information on sharing authentication sessions or data among applications, see Authentication Sharing Strategies

4.1.1 Session Creation

A session starts when an HTTP client makes its first request to a CSP application.

When a new session is created, the CSP server does the following:

1. Creates a new session ID number.
2. Performs licensing checks, as appropriate.
3. Creates a new instance of the %CSP.Session object (which is persistent).
4. Invokes the **OnStartSession** method of the current session event class (if present).
5. Creates a session-cookie in order to track subsequent requests from the HTTP client during the course of the session. If the client browser has disabled cookies, CSP automatically uses URL rewriting (placing a special value into every URL) in order to track sessions.

For the first request of a session, the NewSession property of the %CSP.Session object is set to 1. For all subsequent requests it is set to 0:

```
If (%session.NewSession = 1) {
   // this is a new session
}
```

4.1.2 Session ID

A CSP application can find its particular session ID via the SessionId property of the %CSP.Session object:

```
Write "Session ID is: ", %session.SessionId
```

4.1.3 Session Termination and Cleanup

A session ends for one of the following reasons. (For more information on logging out see the section "Logout or End Session" in this book.)

1. The session times out because it did not receive any requests within the specified session timeout period.
2. The recommended way to logout of a CSP session is to link to the application home page passing a URL that contains the string, `CacheLogout=end`. This ends the current session – releases any license acquired, deletes existing session data, and removes the security context of the session – before it attempts to run the home page.
3. The session is explicitly ended programmatically on the server (by setting the %CSP.Session object's EndSession property to 1. For example, you may wish to end a session if the client is stopped or navigates to a new site.
4. The session can be logged out using the %CSP.Session object's **Logout** method

When a session ends, the CSP server deletes the persistent %CSP.Session object and decrements the session license count, if appropriate. If the session ended because of a timeout or server action, it also invokes the **OnEndSession** method of the session event class (if it is present).

Certain Zen components, notably `tablePane`, store temporary data in `^CacheTemp.zenData`, and this is typically cleaned up automatically by the default Event Class. However, if you define your own custom event class, you must explicitly call **%ZEN.Controller.OnEndSession()** in the **OnEndSession()** callback method in your event class. Otherwise the temp data is not cleaned up.

4.1.4 Reserved CSP Parameters

The table shows reserved parameters and their uses.

Table 4–1: Reserved CSP Parameters

Parameter	Use

Parameter	Use
CacheUserName	From the login page, contains the username to log in; for example, `CacheUserName="fred"`
CachePassword	From the login page, contains the password of the user designated by `CacheUserName`; for example, `CachePassword="fredspwd"`.
CacheOldPassword	If passed in with `CacheUserName` and `CachePassword`, it contains the current password for the user. The security routines changes the user's password to a new value, the one from `CachePassword`, such as, `CacheOldPassword="fredsAboutToBeChangedPwd"`. After the password is changed, the user is logged in using the new password.
CacheRepeatPassword	Not used.
CacheLogin	Not used.
CacheLogout	`CacheLogout` with no value or any value other than "cookie" causes the session for this request to be logged out (but not destroyed.) Logging out destroys the current Login Cookie and removes any two-factor security tokens being held in limbo for this session. `CacheLogout="cookie"` destroys the current Login Cookie.
CacheSecurityToken	`CacheSecurityToken` contains the value of a submitted security token from the Login Security Token page, such as `CacheSecurityToken="12345678"`.
CacheSecuritySubmit	The presence of this name indicates that the user is submitting a security token whose value is associated with `CacheSecurityToken`.
CacheSecurityCancel	The presence of this name indicates that the user has cancelled out of the Login Security Token page.
CacheLoginPage	Login pages, including custom login pages, contain two sub-pages: one for login and one for returning the security token value. The page checks the value of `CacheLoginPage` to determine which subpage to display. `CacheLoginPage=1` indicates the Login subpage should be displayed.

Parameter	Use
CacheNoRedirect	A page P is requested, but it is unauthenticated, so the Login page is displayed. After the user submits the information from the login page, usually the page request for P is redirected back to the browser. (This stops the browser from asking the user to press the <Resend> button before its shows P.) This behavior can be short-circuited by passing CacheNoRedirect=1

4.2 %CSP.Session Object

The %CSP.Session object contains information about the current session as well as a way to control aspects of the session programmatically.

4.2.1 User Session Data — Data Property

You can store application-specific information within the %CSP.Session object using its Data property. Data is a multidimensional array property that lets you associate specific pieces of information in a multidimensional array. The contents of this array are automatically maintained over the lifetime of the session.

You can use the %CSP.Session object Data property in the same way you would use any other ObjectScript multidimensional array.

For example, if the following code is executed within an **OnPage** method:

```
Set %session.Data("MyData") = 22
```

Then a subsequent request to the same session (regardless of which class handles the request) sees this value within the %CSP.Session object:

```
Write $Get(%session.Data("MyData")) // this should print 22
```

The ability to store application-specific data within the %CSP.Session is a very powerful feature but should be used correctly. Refer to the section "State Management" for a further discussion.

4.2.2 Setting User Session Data — Set Command

To store data (only literal data — not object references) in the %CSP.Session object, use the **Set** command. Every node within the Data array can contain a string of up to 32K characters.

```
Set %session.Data("MyData") = "hello"
Set %session.Data("MyData",1) = 42
```

4.2.3 Retrieving User Session Data — Write Command

You can retrieve data from the Data property as part of an ObjectScript expression:

```
Write %session.Data("MyData")
Write %session.Data("MyData",1) * 5
```

If you refer to a node of the Data array that has no value, there is an <UNDEFINED> (undefined) error at runtime. To avoid this, use the ObjectScript **$Get** function:

```
Write $Get(%session.Data(1,1,1)) // return a value or ""
```

4.2.4 Deleting User Session Data — Kill Command

To remove data from the **Data** property, use the ObjectScript **Kill** command:

```
Kill %session.Data("MyData")
```

4.2.5 Session Timeout

CSP sessions automatically track how much time has elapsed since they have received a request from a client. If this elapsed time exceeds a certain threshold then the session automatically times out.

By default, the session timeout is set to 900 seconds (15 minutes). You can change this default for a CSP application in the Management Portal. Navigate to **System Administration** > **Security** > **Applications** > **Web Applications**. Select the application and click **Edit**. You can also set it from within an application by setting the value of the %CSP.Session object AppTimeout property:

```
Set %session.AppTimeout = 3600 // set timeout to 1 hour
```

To disable session timeouts, set the timeout value to 0.

Note that if a session changes CSP applications during its life span, its timeout value will not be updated according to the default timeout defined in the application that the session moved into. For example, if a session starts out in CSP Application A, with a default timeout of 900 seconds, and then moves into CSP Application B, which has a default timeout of 1800 seconds, the session will still timeout after 900 seconds.

If you want an application change to result in the session timeout being updated to that of the new application, use a session event class, override the **OnApplicationChange** callback method, and add code to handle the update of the AppTimeout property of the *%session* object.

4.2.6 Timeout Notification — OnTimeout Method

When a CSP application timeout occurs, the CSP server can notify the application by invoking the **OnTimeout** method of a specified %CSP.SessionEvents class. You can specify the name of this class via the EventClass property of the %CSP.Session object.

By default, there is no event class defined and a timeout simply ends the current session.

4.3 State Management

As HTTP is a stateless protocol. Applications written for the web have to use special techniques to manage the application context or *state*. CSP provides a number of mechanisms for state management. Each of these may be appropriate for specific circumstances.

4.3.1 Tracking Data between Requests

The basic problem of state management within a web application is keeping track of information between successive HTTP requests. There are a number of techniques available for this including:

- Storing data on individual pages using either hidden form fields or URL parameters
- Storing data in cookies on the client
- Storing data in the %CSP.Session object on the server
- Storing data within the Caché database

4.3.2 Storing Data within a Page

To store state information within a page, you must place it so that a subsequent request from this page includes the information.

If the page makes a request via a hyperlink, then the data should be placed within the URL for the hyperlink. For example, here is a hyperlink containing state information defined within a .csp file:

```
<a href="page2.csp?DATA=#(data)#">Page 2</A>
```

When the CSP serves the page containing this link, the expression **#(data)#** is replaced with the value of the server variable data in the text sent to the client. When the user selects this link to page2.csp, the CSP server has access to the value of *DATA* via the *%request* object. If needed, CSP can encode such data. Refer to "Authentication and Encryption" for more details.

If the page contains a form, you can place state information within hidden fields:

```
<form>
<input type="HIDDEN" name="DATA" value="#(data)#">
<input type="SUBMIT">
</form>
```

As with the hyperlink example, when this form is sent to the client, the expression #(data)# is replaced with the value of the variable data. When the user submits this form, the value of *DATA* is available via the *%request* object.

To automatically insert values into all links and forms, use **%response.Context**.

4.3.3 Storing Data in Cookies

Another technique for storing state information is to place it within a cookie. A cookie is a name-value pair stored within the client. Every subsequent request from the client includes all of the previous cookie values.

To set a cookie value, override the page cookie value within the %CSP.Response object:

```
Class MyApp.Page Extends %CSP.Page
{
//...

ClassMethod OnPreHTTP() As %Boolean
{
    Do %response.SetCookie("UserName",name)
     Quit 1
}
}
```

The server can later retrieve this information using the %CSP.Request object's Cookies property.

Storing information within a cookie is useful for information that you want to remember past the end of a session. (To do this, you must set an expiration date as, by default, cookies end when the browser closes.) For example, you could remember a username in a cookie so that in a subsequent session they would not have to reenter this information. See an HTML manual for information regarding different kinds of cookies and their formats.

4.3.4 Storing Data in the Session — Data Property

As discussed in an earlier section, you can store state information in the session %CSP.Session object using its Data property. Any information placed into the *%session* object is available for the remainder of the current session (or until it is removed from the *%session* object).

The *%session* object is a good place to store simple information that is useful across the duration of a session, such as the current user's name. The *%session* object is not good for information that must live beyond the scope of the current session. It is also not a good place for information that is dependent on the navigation path taken by the user through the application. Users are typically free to jump about web applications at will and this can lead to trouble if an application makes assumptions about the specific path taken by a user.

4.3.5 Storing Data in the Database

If you have more complex information to associate with a user, it is probably best to store it within the built-in Caché database. One way to do this is to define one or more persistent classes within the database and store their object ID values within the *%session* object for subsequent access.

4.3.6 Server Context Preservation — Preserve Property

Typically the only processing context preserved by the CSP server from one request to the next is held within the `%session` object. The CSP server provides a mechanism for preserving the entire processing context variables, instantiated objects, database locks, open devices between requests. This is referred to as `context preserving` mode. You can turn context preservation on or off within a CSP application at any time by setting the value of the %CSP.Session object Preserve property. Note that tying a process to one session results in a lack of scalability.

4.4 Authentication and Encryption

It is fairly common to place state information on pages sent to the HTTP client. When subsequent requests are made from these pages, the state information is sent back to the server. Many times, it is important that state information be placed on a web page in such a way that a) viewers of the HTTP source cannot determine the value of the state information and that, b) the server can verify that the returning information was, in fact, send out from the same server and session. Via its encryption services, CSP provides an easy-to-use mechanism to accomplish this.

4.4.1 Session Key

CSP can encrypt and decrypt data on the server using an encryption key. Every CSP session has a unique session key (accessible via the %CSP.Session object Key property) that is used to encrypt data for a session. This mechanism is secure because the session key is never sent to an HTTP client; it remains on the CSP server as part of the %CSP.Session object.

You can manually encrypt values on the server using the **Encrypt** method of the %CSP.Page class. You can subsequently decrypt this value using the **Decrypt** method.

4.4.2 Encrypted URLs and CSPToken

In certain circumstances (described below) a class generated from a .csp file automatically encrypts URL values sent to the client (for manually created classes, you must invoke the **Link** method of the %CSP.Page class to perform this action).

For example, suppose a .csp file contains an anchor tag defining a link to another page:

```
<a href="page2.csp?PI=314159">Page 2</a>
```

If this URL is encrypted, the following may be sent to the client:

```
<a href="page2.csp?CSPToken=8762KJH987JLJ">Page 2</a>
```

When the user selects this link, the encrypted parameter *CSPToken* is sent to the CSP server. The server then decrypts it and place its decrypted contents into the *%request* object. If the encrypted value has been modified or sent from a different session then the server throws an error. You can use the %CSP.Request class **IsEncrypted** method to determine whether a parameter value was originally encrypted.

The CSP compiler automatically detects all the places where a URL can occur within an HTML document and performs encryption as needed (based on the class parameters of the target page as described in the following section). If you are creating a page programmatically you can get the same behavior using the **Link** method of the %CSP.Page class.

If you are passing a link as an argument to a function, always use the **Link** method of the %CSP.Page class, rather than the **#url()#** directive, as in the following example:

```
window.showModalDialog('#(..Link("locks.csp"))#','',windowFeatures);
```

This example of using **#url()#** as an argument to a function *does not work*:

```
window.showModalDialog('#url(locks.csp)#','',windowFeatures);
```

If you need to provide an encrypted URL within a .csp file in a place that is not detected by the CSP compiler, use the **#url()#** directive. For example, in a client-side JavaScript function, where the link is a parameter, you can use:

```
<script language=JavaScript>
function NextPage()
{
    // jump to next page
    CSPPage.document.location = '#url(nextpage.csp)#';
}
</script>
```

4.4.3 Private Pages

CSP provides the notion of a *private* page. A private page can only be navigated to from another page within the same CSP session. Private pages are useful for applications where you want to restrict access to certain pages.

For example, suppose there is a private page called private.csp (one of the CSP sample pages). A user cannot navigate directly to private.csp (for example, by typing in its URL). A user can only navigate to private.csp from a link contained within another CSP page. The link contained in the referring CSP page cannot be an absolute URL, starting with `http://`. Only paths relative to the referring page are properly encrypted/tokenized by the private pages method. That is: The first two links below pass the same token to the target private page, test2.csp.

```
<A HREF='test2.csp'>Link to private page - relative path</A> <BR>
<A HREF='/csp/samples/test2.csp'>
       Link to private page - full application path</A> <BR>
```

This link is hashed differently and fails access.

```
<A HREF='http://myserver/csp/samples/test2.csp'>
        Link to private page - absolute path</A>
```

The user also cannot bookmark a private page for later use because the encrypted token used to protect the private page is only valid for the current session.

Private pages work as follows. The %CSP.Page subclass responsible for the page has its class parameter PRIVATE set to 1. A URL requesting this page must contain a valid, encrypted *CSPToken* value in its query string. Any links to this page processed by CSP automatically have an encrypted *CSPToken* value.

4.4.4 Encoded URL Parameters

In a manner similar to private pages, you can specify that the URL parameters of a CSP page are to be encoded by setting the value of the %CSP.Page class parameter ENCODED. ENCODED can be set to 0, 1, or 2. Any links to a page whose ENCODED class parameter is 1 or 2 automatically have any URL parameters encoded within the encrypted *CSPToken* value. If ENCODED is set to 2, then values must be encoded; if 1, it is possible to mix encoded and unencoded values.

The three settings for ENCRYPTED are:

- ENCODED=0 — Query parameters are not encrypted
- ENCODED=1 — Query parameters are encrypted and passed within *CSPToken*
- ENCODED=2 — Same as '1' except any unencrypted parameters are removed from the *%request* object before calling the Page method. This ensures that only encrypted parameter are available in the %CSP.Request object.

Note that because ENCODED=2 removes unencrypted parameters from the url it can disable components such as the Zen <form> element.

4.4.4.1 Example of ENCODED=2

For example, suppose you have two .csp pages. One page (list.csp) displays a list of bank accounts as hyper-links and a second page (account.csp) displays information about a specific account. account.csp expects a URL parameter named *ACCOUNTID* to determine which account to display. We do not want to publish account numbers on the client and we do not want unauthorized access to account.csp or the ability to display any other account number. We can do this by setting the account.csp ENCODED parameter to 2. Here are the relevant .csp files:

Source for list.csp

```
<html>
<body>
Select an account:<br>
<a href="account.csp?ACCOUNTID=100">Checking</a>
<a href="account.csp?ACCOUNTID=105">Saving</a>
</body>
</html>
```

Source for account.csp

```
<html>
<csp:class private=1 encoded=2>
<body>
Account Balance: <b>$#(..GetBalance())#</b>
</body>

<script language="Cache" method="GetBalance" arguments=""
        returntype="%Integer">
    // server-side method to lookup account balance
    New id
    Set id = $Get(%request.Data("ACCOUNTID",1))
    If (id = 100) {
        Quit 157
    }
    ElseIf (id = 105) {
        Quit 11987
    }
    Quit 0
</script>

</html>
```

The CSP server sends the following HTML to the client when list.csp is requested:

```
<html>
<body>

Select an account:<br>
<a href="account.csp?CSPToken=fSVnWw0jKIs">Checking</a>
<a href="account.csp?CSPToken=1tLL6NKgysXi">Saving</a>
</body>
</html>
```

Notice that only the encrypted values for ACCOUNTID are sent to the client.

When account.csp is processed, it sees the decrypted value for *ACCOUNTID* (referenced in its **GetBalance** method).

4.4.4.2 Example of ENCODED=1

The difference between ENCODED=2 and ENCODED=1 is that with a ENCODED=2, any text that is added to the URL in an unencrypted form is thrown away. If ENCODED=1 is used, then the unencrypted text is passed through to the page. The page can then include code that specifies what to do with this unencrypted text.

The samples pages protected.csp and protectedentry.csp show an example of using ENCODED=1. The source for `protected.csp` shows that it checks to see if anything has been added to the URL that is not encrypted. If there is anything that is not encrypted, the page displays a scrolling marquee that says `HACKER ALERT!`

To see this, go the samples page.

1. Double-click protectedentry.csp.
2. Enter 500 in the **BALANCE:** field
3. Click **View Balance**
4. The `protected.csp` page is displayed saying `Your Account Balance is: 500`
5. Notice that the URL contains an encrypted CSPToken (everything after CSPTaken=). This is the 500 that you entered encrypted.
6. Navigate to the end of this URL and enter `&BALANCE=8000` and press Enter.
7. The `protected.csp` page is displayed. It accepted the unencrypted addition to the URL and acted on it by displaying the `HACKER ALERT!` marquee. If ENCODED had been set to 2, it would have ignored the unencrypted entry.

4.5 Authentication Sharing Strategies

This section describes how to create a set of applications to work as a group in two ways:

- Sharing authentication: If applications do not share authentication, the user must log in to each application that is linked to by another application separately. Shared authentication allows the user to enter all linked applications with a single login.
- Sharing data: The applications may want to share and to coordinate global state information.

This section describes the following:

- Authentication Approaches
- Authentication Architecture
- Considerations when Choosing Your Strategy

4.5.1 Authentication Approaches

This section describes the following approaches to authentication. Options to implement them are found in the Management Portal by navigating to **System Administration** > **Security** > **Applications** > **Web Applications**.

- One-Time Sharing: Login Cookies. All applications with the same id share authentication. This corresponds to the CSP Application option **Login Cookies**.
- Continuous Sharing: Authentication Groups By ID or Session
  - By Session. This corresponds to two CSP Application options. The **Session Cookie Path** option makes applications with the same session-path-cookie share a session and its authentication. If the **CSPSHARE** option is set as CSP-SHARE=1, then when the user clicks a link (in a source application) to a target application, then the target application is placed in the same session as the source application

4.5.1.1 One-Time Sharing: Login Cookies

Login Cookies hold information about the most recently logged-in user. If you want to keep your users from having to log in too often, but you want your applications to remain distinct and unconnected, use Login Cookies.

For Login Cookies, place each application in a separate session. Then authentication is shared only when an application is entered for the first time. Login Cookies applications do not form a group. So after login, changes in authentication in one application do not affect the other applications.

When a user logs in with a password, that authentication is saved in a cookie. If another application with Login Cookies enabled is entered (for the first time), it uses the authentication saved in the cookie. If the user jumps to a third application (for the first time) which does not have Login Cookies enabled, the user must enter a username/password.

See the section "Considerations in Choosing Your Strategy" for more information for deciding what strategy to use.

When deciding whether or not to use Login Cookies, here are some considerations:

- The login cookie is updated to a new user whenever the user logs in with a password.
- Login cookies are not generated for an unauthenticated login (as UnknownUser).
- Login cookies are not generated when logging in through API calls.
- Login cookie sessions are independent once that session has been authenticated. So logging out or timing out in one session does not affect the other sessions.
- Authentication from a Login-Cookie application cannot be shared with a password-only (non-Login-Cookie) application. For authenticated applications in a group, for consistent behavior, use Login Cookies for all or for none.

4.5.1.2 Continuous Sharing: Authentication Groups By ID or By Session

With group sharing, the authentication of the group's applications moves as a unit. If a user in an application in a group logs in as a new user, all the applications move to that user. If one application logs out, they are all logged out.

Applications can be grouped together in two ways: By Session and By ID. By-Session groups share authentication and data. By-ID groups share authentication only.

Applications are run in CSP sessions. Each session has a security context associated with it. For more on CSP sessions, see the section About CSP Sessions in this book.

If several applications are placed in the same session, they share authentication. This is called a By-Session group (session-sharing). In addition, the session may contain user defined data. Applications can be made to share a session by having their Application Cookie Path match exactly or by using the CSPSHARE=1 flag when jumping via a link from one application to another.

Applications can be grouped by assigning the applications matching group identifiers. This is called a By-ID group. The group shares a security context. The applications are usually in separate sessions. The group does not manage user data, only authentication.

See the section "Considerations in Choosing Your Strategy" for more information.

4.5.1.3 By-Session Groups (Session-Sharing)

Sharing a session has potential issues. Session events are picked up only from the original CSP application. If the link goes to a page that requires different session events, then these session events do not run. Also, running a page in another CSP application with a different security context may require a login; the login might alter the security context of running pages in the original CSP application. Before choosing to use By-Session groups, please read Considerations in Choosing Your Strategy below.

When applications share a session, they share both authentication and data via the session object. There are two ways to share a session:

1. Session Cookie Path: All applications with exactly-matching session cookie paths are placed into the same session.
2. CSPSHARE: Putting CSPSHARE=1 in the link to the application page. Use this when the source application's Session Cookie Path is different from the target's Session Cookie Path.

If By-Session sharing is required, then the best solution is to name all applications so they can be given the same Session Cookie Path. You may have to rename your applications because the Session Cookie Path must be a substring of the application name.

If this cannot be done and session sharing is required, then you have to put the CSPSHARE parameter in links that jump from one application to another. The target application page is placed in the same session as the source application's pages. The source's session is determined either from the CSPCHD parameter or the session cookie.

See the section "Considerations in Choosing Your Strategy" for more information.

4.5.1.4 By-ID Groups

You can group your applications by navigating to **System Administration** > **Security** > **Applications** > **Web Applications** on the Management Portal and giving them a group name in the **Group by Id field**. This name groups opened applications together. Groups are in different sessions. The applications do not share data.

The group name is attached to an application, not a namespace. Applications with the same group name share authentication regardless of namespace.

Authentication is shared within a single browser only.

See the section "Considerations in Choosing Your Strategy" for more information.

4.5.1.5 CSPSHARE

When CSP receives a request from a browser it does a series of checks to see if the sessionId it receives is a valid one. These checks include:

- Whether the User-Agent is the same as that of previous requests from this sessionId
- If cookies are being used, whether this sessionId comes from a cookie or from a CSPCHD parameter

If you pass a `CSPSHARE=1` query parameter, CPS turns off this checking. Then you can construct a link to another CSP application and include the current sessionId, using `CSPCHD=`*sessionId*, so that this link runs in the same session as your existing page. In addition, if `CSPSHARE=1` when you construct a link, CSP automatically inserts the `CSPCHD=`*sessionId* in to the link. If you manually insert a link with Write statements, you may need to insert the sessionId manually.

For example, if you have an application that requests an `http` page from an `https` page (or an `https` page from an `http` page), add `CSPSHARE=1` to the link as follows:

```
#(..Link(%request.URL_"?CSPSHARE=1"))#
```

`CSPSHARE=1` forces the link construction to add CSPCHD to share the sessionId even if Caché detects that cookies are enabled.

See the section "Considerations about CSPSHARE" for more information.

4.5.2 Authentication Architecture

4.5.2.1 Security Context & Sticky Logins

Applications are run in sessions. A session requires a security context in which to run an application. The security context contains the authentication state.

By-Sessions and By-ID Groups have a *sticky login* which remembers the security context of the last application used in the session or group. If a user in a group application logs in as a different user, the sticky login is updated. (The sticky login is not updated if the user logs in to an unauthenticated application.)

When jumping to an application in a session, the session attempts to use the sticky login appropriate for the target application. If the sticky login does not match the session's current security context and the application can accept the authentication method in the sticky login, the session's security context is switched to that in the sticky context.

A session's sticky login is lost when the session is ended. The group's sticky login is lost when all the sessions containing any of the group's applications are ended.

After the initial login, a group has an associated sticky login object which it attempts to use when entering one of the group's applications. The sticky login is not updated when an application in the group is entered as UnknownUser as this would have the effect of moving all other applications in the group to the unauthenticated security context.

If the sticky login contains a two-factor authenticated user, that two-factor authentication is used for non-two-factor applications, so long as the username authentication matches in the two applications.

4.5.2.2 Cascading Authentication

The CSP Server uses precedence when attempting to obtain authentication information for an application. It attempts to get new authentication information in each of the following events:

- For the first request to a new session;
- When there is an application change within the session;
- When the application is part of a By-id group and the session's current security context does not match that of the group's sticky context;
- When the request contains a username/password pair.

It attempts to get new authentication information sequentially in the following order:

1. Explicit Login: Checks to see if the user entered an authenticated username/password. If they did, the system updates the application's authentication group's context. (This sets the group's Sticky Login.)
2. Sticky Login: Get the Application's group's sticky context. If no sticky login and group-by-session, use session's current context.
3. Login Cookie: Use if one exists and is enabled for this application.
4. Unauthenticated: Use Unknown User if enabled for application.

5. Put up Login Page: If all the above fail, then request username/password from user. If called from the %CSP.Session API, then only username/password is tried. After login, update the group's sticky login unless just logged in as UnknownUser.

4.5.2.3 Logout or End Session

Authentication is lost when a session is logged out or ended. You can use the following %CSP.Session methods to logout or end a session:

Recommended: CacheLogout=end

The recommended way to logout of a CSP session is to link to the application home page passing a URL that contains the string, `CacheLogout=end`. This ends the current session – releases any license acquired, deletes existing session data, and removes the security context of the session – before it attempts to run the home page.

If this CSP application requires authentication, there is no session and no authenticated user. In this case, Caché does not run the home page logic but displays the login page instead. When the user submits a valid login this starts this new session and then displays the home page.

Set EndSession? =1

This kills the session. The session's sticky context is destroyed. **OnEndSession** is called. If the session contains a By-Session group, then the group is destroyed. If the session contains a By-Id application, then that application is removed from the group which continues to exist unless this was the only application in the group. Login cookies are unaffected. By-Session groups lose their data. However, for By-Id groups, the sticky-login for the group is unaffected by a singular destruction and the other members of the group remain logged in.

In addition, for By-Session groups, the destruction *disperses* the members of the group and if the member applications are reentered, it cannot be guaranteed that they will be reintegrated into the same new session or (if they were grouped using CSPSHARE) sent to diverse sessions.

Session Logout

The session is logged out. Its sticky context is destroyed. If the session contains a by-session group, then all the applications in the group lose their authentication. If the session contains an application from a by-id group, then group loses its sticky context and all the applications in the group are logged out.

In addition, **OnLogout** is called. The login cookie is destroyed.

The session continues to exist, so data is retained for By-Session groups.

Session Logout All

It is possible to log out all session currently authenticated as a particular user.

This zaps the login cookie.

The sessions continue to exist but have not authentication.

4.5.3 Considerations in Choosing Your Strategy

This section contains some points to consider when you are choosing your strategy. See the section "One Time Sharing: Login Cookies" for more information for deciding what strategy to use.

4.5.3.1 Considerations for Groups

This section contains some points to consider when you are creating authentication groups.

1. Use session-sharing only when you decide that data must be shared via the session object. By-ID and Login Cookies-sharing are more robust and predictable.

2. When creating groups, be as consistent as possible to create uniform behavior for your targeted users. Do not place an application in both a By-ID group and a By-Session group. Using the different authentication strategies may cause unexpected behavior. By-ID takes precedence over By-Session. So if an application has both, it stays synchronized By-ID.
3. Use the same authentication types for all members of the group. In particular, if some applications in the group allow Login Cookies and others do not, then entering the group via a username/password authenticates the entire group, whereas entering it via a login cookie authenticates only some of the applications. This can cause confusion among your users about why sometime a login is required and other times not.
4. The CSP server considers every application to be in an Authentication Group. A lone application in a session forms a single-entity By-Session authentication group.)
5. Try not to put unauthenticated-only applications in By-ID groups.
6. By-Session groups are fragile; using By-ID is a more robust approach. Since all the information about a group, including its shared data, is contained in a single session, the group can easily be lost. This is because a session can time-out, that is, after a specific amount of time the session is automatically destroyed. If the user steps away from his computer or uses an application which is not in the By-Session group, the session may timeout. If one of the applications in the group marks ENDSESSION=1, the group is dispersed.
7. If the browser has open tabs containing pages from the dispersed applications, clicking on them may require multiple logins, especially if they were originally grouped using CSPSHARE=1. In any case, the data from the original session is permanently gone.
8. When a group loses its authentication, refreshing or going to an open page from a group application requires that the user re-login.
9. Ending a session containing a By-Session application requires that the user re-login when refreshing any page of any application in that by-session group. Killing a session containing a By-ID application does not require any logins unless that session's application was the only member of the group.
10. Logging out a session logs out all members of the session's group, even if they are in different sessions. Refreshing any of the group's pages requires a new login. However, for By-ID groups, one login logs in the entire group. For By-Session groups, one login logs in the entire group as long the CSP Gateway is able to direct the dispersed applications back to a newly constructed session object.
11. Logging out does not destroy the session, so any session data continues to exist.
12. One cannot have same application logged in to two different users in different tabs of the same browser.
13. Authentication is shared within a single browser only. This runtime identifier is stored in the %Session object.
14. Grouping allows you to share authentication with users that are in the same group (By-ID) or the same session (By-Session). If you want to share authentication from applications that are outside your specified group, use Login Cookies. If you want to send authentication to applications outside your specified group, use CSPSHARE=1. (See the section “Considerations about CSPSHARE” in this book.)

4.5.3.2 Considerations about CSPSHARE

Use CSPSHARE as a last resort.

By-Session application links do not need CSPSHARE=1 in the following cases:

- If the source and target applications have the same group ID.
- If the target page is in the same application as the source page.
- If the target page application's Session Cookie Path matches the source application's Session Cookie Path.

4.5.3.3 Sharing Data

By-Session groups can share data via the session object.

By-ID groups must manage their own data. If the data is stored, for example, in a global, the data could be keyed using the current user, $Username, or by the group's runtime ID. The CSP Server assigns each browser a browser-id cookie. When a By-Id group is created it is assigned a key which is the browser ID concatenated with the group ID. This creates a unique key, %CSP.Session.BrowserId, which can be used as a key under which to store data.

5

Tag-based Development with CSP

CSP lets you develop CSP applications using standard HTML files. The CSP compiler converts HTML (and XML) documents into %CSP.Page classes that can respond to HTTP requests.

Classes generated by the CSP compiler are no different from, and are completely interoperable with, classes that you create yourself. This gives you the choice between developing CSP pages through HTML pages or through creating classes that are extensions of the %CSP.Page, which can be used within applications. Examining the generated CSP classes is often useful in debugging.

The HTML documents processed by the CSP compiler contain tags that might, for example, control class generation, provide control flow, manage data access, and control server-side behavior. These tags are the *CSP markup language* or *CSP tags*. These tags are interpreted on the CSP server at development time. The HTML sent to the HTTP client by CSP is completely standard and includes no CSP tags.

Within a CSP file, you can use normal HTML tags plus:

- Caché data expressions using **#( )#**, which substitute values at page generation.
- Caché CSP tags, `<csp:xxx>`, which provide built-in and custom functionality.
- Caché scripts, `<script language=cache runat=server/compiler>`, which execute Caché code during page generation or page compilation.
- Caché methods: reusable class methods that you can invoke from within a page.
- Server-side subroutine calls, **#server()#** and **call()#**, which invoke server-side subroutines from client-side code (hyperevents).
- Custom tags, described in the chapter "Developing Custom Tags"

5.1 CSP Compiler

The CSP compiler is a set of Caché classes and programs running on a Caché server that

1. Reads and parses an HTML document using the CSP markup language,
2. Applies pattern matching logic based on CSP rules,
3. Generates a Caché class, and
4. Compiles the class into executable code.

For example, when the following simple CSP document, hello.csp is compiled,

```
<html>
<body>
Hello!
</body>
</html>
```

the CSP compiler transforms this into a class similar to:

```
Class csp.hello extends %CSP.Page
{

ClassMethod OnPage() As %Status
{
    Write "<html>"
    Write "<body>"
    Write "Hello!"
    Write "</body>"
    Write "</html>"
    Quit $$$OK
}
}
```

When a user requests the hello.csp page from a browser, the CSP server invokes the generated **OnPage** method and the original text of the CSP document is sent to the browser for display.

5.1.1 Automatic and Manual Page Compilation

You can have the CSP server compile CSP source documents into classes either automatically or manually.

If `autocompile` mode (the default) is on, the CSP server automatically asks the CSP compiler to compile CSP source documents into classes as needed. The CSP server compares the timestamp of the source files with the class timestamp and recompiles any page whose source is newer than its class. Typically this mode is turned off in deployed applications to avoid the overhead of checking the timestamps:

To turn off `autocompile`,

1. In the Management Portal, navigate to **System Administration** > **Security** > **Applications** > **Web Applications**.
2. Select an application in the table and click **Edit**.
3. On the **Edit CSP Application** page, clear **Autocompile**.

You can explicitly compile a CSP source file into a class. This is useful for finding errors.

1. Open the CSP source file in Studio.
2. Select **Build > Compile**.

You can also compile a CSP source file from the Caché command line (the terminal) using the **$System.CSP** API (as shown in the example). This method loads and compiles the CSP file with the URL path (not physical path) /csp/user/mypage.csp. The `c` (*c*ompile) flag compiles the generated class. The `k` flag (*k*eep) preserves the generated intermediate code to be viewed.

```
Do $System.CSP.LoadPage("/csp/user/mypage.csp","ck")
```

5.2 CSP Markup Language

The CSP markup language is a set of directives and tags you can use to control the classes generated by the CSP compiler.

When you compile a CSP document, the result is a Caché class that executes ObjectScript or Basic code. Keep this in mind to help you to develop correct application logic as well as perform troubleshooting. You may, in fact, find examining the code generated by the CSP compiler a useful way to learn more about both CSP and CSP markup language.

It is also important to keep track of what code is executed on the CSP server (as it prepares a response to an HTTP request) and what code is to be executed on the HTTP client (such as HTML and JavaScript).

5.2.1 CSP Page Language

By default, the CSP compiler evaluates runtime expressions and generates code using ObjectScript. For a given CSP document, you can change this default to Basic by putting the PAGE directive at the top of the document:

```
<%@ page language="Basic" %>
```

For an example, see the basic.csp application included in the CSP samples (click **source** to view the source).

Within a CSP document, runtime expressions as well as the contents of any server-side <script> tags must use the default language for the page (or else you receive a compile-time error). Or you can define a method in a different language and invoke that from the default language.

5.2.2 Text

Any text contained within a CSP document (HTML or XML) that is not a CSP directive or special tag is sent unchanged to the HTTP client requesting the page.

For example, a CSP document containing the following:

```
<b>Hello!</b>
```

generates the following code within the generated class:

```
 Write "<b>Hello!</b>",!
```

which, in turn, sends the following to the HTTP client:

```
<b>Hello!</b>
```

5.2.3 Compile-time Expressions and Code

You can specify that an expression be evaluated at compilation time (as opposed to runtime) of the CSP page. Such expressions are typically used in CSP Rule definitions, though there are times when they may be of use elsewhere.

Compile-time expressions are delimited using the **##(expr)##** directive, where *expr* is an ObjectScript expression.

For example, a CSP document containing the following:

```
This page was compiled on: <b>##($ZDATETIME($H,3))##</b>
```

generates the following code in the generated class:

```
 Write "This page was compiled on <b>2000-08-10 10:22:22</b>",!
```

You can also define lines of code to be executed at page compilation time using the *runat* attribute of the `<script>` tag:

```
<script language="Cache" runat="compiler">
```

Note: You must write all *compile-time* expressions and code using ObjectScript.

5.2.4 Runtime Expressions

A CSP document may contain expressions that are run on the CSP server when the page is served (that is, at runtime). Such expressions are delimited using the **#(expr)#** directive, where *expr* is a valid ObjectScript or Basic expression (depending

on the default language for the page; the language used within a runtime expression must match the default language for the CSP document.)

Note: Note that name indirection is supported and argument indirection is *not* supported with the **#(expr)#** directive.

For example, a CSP document containing the following:

```
Two plus two equals <b>#(2 + 2)#</b>
```

generates the following code within the generated class:

```
Write "Two plus two equals <b>", (2 + 2), "</b>",!
```

which, in turn, sends the following to the HTTP client:

```
Two plus two equals <b>4</b>
```

Samples of runtime expressions;

- Value of a variable set earlier on the page

  ```
  The answer is <b>#(answer)#</b>.
  ```

- Object property or method

  ```
  Your current balance is: <b>#(account.Balance)#</b>.
  ```

- Field within a %ResultSet object

  ```
  <table>
  <csp:while condition="result.Next()">
  <tr><td>#(result.Get("BookTitle"))#</td></tr>
  </csp:while>
  </table>
  ```

- URL parameter using the `%request` object

  ```
  <table bgcolor='#(%request.Data("tablecolor",1))#'></table>
  ```

A runtime expression can be anywhere within an CSP document where the **#(expr)#** structure can be used as legal HTML. This includes within HTML text, as the value of an HTML element attribute, or within the definition of client-side JavaScript code.

If the value of a runtime expression contains any special characters (such as < or > , angle brackets) you need to escape them to send the correct escape sequence to the HTTP client. To escape them, use one of the escape methods provided by the %CSP.Page class. See "Escaping and Quoting HTTP Output" for details. The example below shows the **EscapeHTML** classmethod. Any characters that need to be escaped that exist in *object.Description* are replaced by their HTML escape sequences when the method is run.

```
Description: <b>#(..EscapeHTML(object.Description))#</b>.
```

If a runtime expression is used in an HTML attribute value, any HTML entities found in the runtime expression are converted to the characters that they represent before being converted to executable code. For example:

```
<font size=#(1 &gt; 0)#>
```

generates the following code within the generated class:

```
 Write "<font size=",(1 > 0),">",!
```

5.2.5 Runtime Code

If you need more than a simple expression to run on the CSP server within a page, you can place lines of code to run on the CSP server using the <script runat=server> tag. As with runtime expressions, you can use runtime code for a variety of purposes. The language used for runtime code (specified by the LANGUAGE attribute of the <script> tag) must match the default language for the CSP document.

For example, a CSP document containing the following:

```
<ul>
<script language="cache" runat=server>
    For i = 1:1:4 {
        Write "<li>Item ",i,!
    }
</script>
</ul>
```

generates the following code within the generated class:

```
 Write "<ul>",!
 For i = 1:1:4 {
    Write "<li>Item ",i,!
 }
 Write "</ul>",!
```

which, in turn, sends the following to the HTTP client:

```
<ul>
<li>Item 1
<li>Item 2
<li>Item 3
<li>Item 4
</ul>
```

5.2.6 Runtime Code ObjectScript Single Line

You can use the following syntax to run a single line of ObjectScript. This works for only a single line. The line cannot wrap.

```
#[ set x = a + b write x ]#
```

5.2.7 Server-Side Method

In a CSP document, you can define a method that belongs to the class generated for the document. This is done using arguments with the <script> tag.

You can specify the name of the method as well as its argument list and return type. You can specify the language used to implement the method; this language does *not* need to match the default language for the CSP document.

For example, the following defines a method called **MakeList** that creates an ordered list containing *count* items:

```
<script language="Cache" method="MakeList"
arguments="count:%Integer" returntype="%String">
   New i
   Write "<ol>",!

   For i = 1:1:count {
      Write "<li> Item",i,!
   }
   Write "</ol>",!
   Quit ""

</script>
```

You can then invoke this method from elsewhere within the CSP document:

```
<hr>
#(..MakeList(100))#
```

You can also use inheritance (using the <csp:class> tag) to inherit previously defined methods into your page class or to invoke the class methods of another class:

```
<hr>
#(##class(MyApp.Utilities).MakeList(100))#
```

5.2.8 SQL <script> Tag

You can use SQL to define a Caché %ResultSet object within a CSP page using the following `<script>` tag.

The following example creates an instance of a dynamic SQL %ResultSet object named `query`, prepares the specified SQL query, and executes it (gets it ready for iterating over it).

```
<script language="SQL" name="query">
SELECT Name FROM MyApp.Employee ORDER BY Name
</script>
```

Typically you use a %ResultSet object created by the SQL script tag in conjunction with the <csp:while> tag (see csp:while Tag) to display the results of the query.

The SQL `<script>` tag closes the instantiated %ResultSet object when the page is done executing.

You can specify parameters for the SQL query by using the `?` character within the SQL text. You can provide values for parameters using the *P1*, *P2*,...*Pn* attributes of the SQL `<script>` tag (where *n* is the number of parameters).

Here is example using the SQL `<script>` tag to display the purchases made by the current user. (The current user's User ID is assumed to have been previously stored in the *%session* object):

```
<script language=SQL name=query P1=#(%session.Data("UserID"))#>
SELECT DateOfPurchase,ItemName,Price
FROM MyApp.Purchases
WHERE UserID = ?
ORDER BY DateOfPurchase
</script>
<hr>
Items purchased by: <b>#(%session.Data("UserID"))#</b>
<br>
<table>
<tr><th>Date</th><th>Item</th><th>Price</th></tr>
<csp:while condition="query.Next()">
<tr>
<td>#(..EscapeHTML(query.GetData(1)))#</td>
<td>#(..EscapeHTML(query.GetData(2)))#</td>
<td>#(..EscapeHTML(query.GetData(3)))#</td>
</tr>
</csp:while>
</table>
```

Using the <csp:query> tag, you can use a query defined as part of a Caché class to create a %ResultSet object:

```
<csp:query NAME="query" CLASSNAME="Sample.Person" QUERYNAME="ByName">
```

You can use the resulting %ResultSet object in the same way as you would with the SQL`<script>` tag.

5.2.9 Controlling the Generated Class

Using the <csp:class> tag, you can exert some control over the class generated by the CSP compiler. This control includes selecting superclasses for the class and defining values for many of the %CSP.Page class parameters.

For example, suppose that, in addition to inheriting from the usual %CSP.Page class, you would like a generated class to also inherit from another class. The *SUPER* attribute takes a comma-delimited list of classes and uses them as the superclasses for the generated class.

```
<csp:class SUPER="%CSP.Page,MyApp.Utilities">
```

Here is an example of redefining the value of a class parameter: To redefine the value of the class parameter *PRIVATE* as `1` (to define a page as private), use:

```
<csp:class PRIVATE=1>
```

5.3 Control Flow

The CSP markup language provides several tags to facilitate control over the execution of pages. While not as general purpose as straight server-side tags, these tags can make certain tasks easy to accomplish.

5.3.1 <csp:if> Tag

The <csp:if> tag, along with the <csp:else> and <csp:elseif> tags, provides a way to define conditional output in a CSP page.

The `<csp:if>` tag has a single attribute, *condition*, whose value is an ObjectScript or Basic expression (depending on the default language specified for the page) evaluated by the CSP server at runtime. If this value is true, then the contents of the tag are executed.

For example:

```
<csp:if condition='user="Jack"'>
Welcome Jack!
<csp:elseif condition='user="Jill"'>
Welcome Jill!
<csp:else>
Welcome!
</csp:if>
```

5.3.2 <csp:while> Tag

The <csp:while> tag provides a way to repeatedly process a section of a CSP document as long as a given server-side condition is true.

The `<csp:while>` tag's *condition* attribute contains an ObjectScript or Basic expression (depending on the default language for the page) which is evaluated on the CSP server when a page is served. As long as the condition evaluates to true (1), the content of the `csp:while` tag is evaluated.

The `<csp:while>` tag is typically used with a Caché %ResultSet object to display the results of an SQL query in HTML. In the example below, the contents of the `<csp:while>` tag, which writes out the value of the query's Name column, is repeatedly executed until the %ResultSet object's **Next** method returns the value `FALSE (0)`, indicating the end of the result set.

```
<script language=SQL name=query>
SELECT Name
FROM MyApp.Employee
ORDER BY Name
</script>

<csp:while condition="query.Next()">
#(..EscapeHTML(query.Get("Name")))#<BR>
</csp:while>
```

Using the `<csp:while>` tag's `counter` attribute you can define a counter variable that is initialized to zero (0) and automatically incremented by one (1) at the start of every iteration.

Here, for example, is a way to use the `<csp:while>` tag to create an HTML table with 5 rows:

```
<table>
<csp:while counter="row" condition="row<5">
<tr><td>This is row #(row)#.</td></tr>
</csp:while>
</table>
```

Here is an example of using a not operator (a single quote) in a condition. Note that the condition cannot contain any spaces and does not include start and end quotes. You can also state the condition using parentheses, as `(mystr'="QUIT")`.

```
<csp:while condition=mystr'="QUIT">
//add code
</csp:while>
```

5.3.3 <csp:loop> Tag: Numbered List Example

The <csp:loop> tag provides another way to repeatedly execute content in a CSP document.

The `<csp:loop>` tag lets you define a counter variable (using its *counter* attribute) as well as its starting, ending, and, increment-by value. The default increment-by value is 1.

For example, you can use the `<csp:loop>` tag to create a list containing 5 items:

```
<ul>
<csp:loop counter="x" FROM="1" TO="5">
<li>Item #(x)#
</csp:loop>
</ul>
```

5.4 Escaping and Quoting HTTP Output

To create the literal display of special characters used in HTML, you have to use escape sequences. For example, to display the > (right angle bracket) character in HTML, which has a special meaning in HTML, you have to escape it using the sequence of characters:`>`. Different parts of a CSP document may use different escaping rules (such as HTML and JavaScript).

The %CSP.Page class provides a number of escaping and quoting methods:

- An escaping method takes a string as input and returns a string with all special characters replaced with the appropriate escape sequences.
- A quoting method takes a string as input and returns a quoted string (that is with the appropriate enclosing quotation marks). The quoted string also has all special characters replaced with escape sequences.
- For every escaping method, there is a corresponding unescape method that replaces escape sequences with plain text.

5.4.1 Escaping HTML with EscapeHTML

The %CSP.Page class can replace characters with their corresponding HTML escape sequences.

For example, if a CSP file needs to display the value of a server-side variable, *x*, on a browser, any characters within *x* can be escaped with this expression:

```
#(..EscapeHTML(x))#
```

If the value of *x* is `<mytag>`, the following escaped text is sent to the HTTP client:

```
&lt;mytag&gt;
```

Similarly, escaping is necessary when you send the value of HTML attributes:

```
<input type="BUTTON" value="#(..EscapeHTML(value))#">
```

If the value of *value* is <ABC>, this results in the following text being sent to the HTTP client, where the two angle brackets, left and right, are replaced with their character sequence equivalents: `<` and `>` respectively:

```
<input type="BUTTON" value="&lt;ABC&gt;">
```

Place "" (quotation marks) around the **#()#** directive to make the resulting HTML attribute value quoted.

When sending output from a database to an HTTP client, it is good practice to escape it. For example, consider the following expression that writes a username to a web page (assuming *user* is a reference to an object with a Name property):

```
User name: #(user.Name)#
```

If your application lets users enter their names into the database, you may find that a mischievous user may enter a name containing HTML commands. If the following is written out to an HTTP client without HTML escape sequences, the page may have unintended behavior.

```
Set user.Name = "<input type=button onclick=alert('Ha!');>"
```

5.4.2 Escaping URL Parameters with EscapeURL

Parameter values in URL strings can also be escaped. URLs use a different set of escape sequences than HTML. The %CSP.Page class **EscapeURL** method replaces all special URL parameter value processing characters with their corresponding escape sequences.

For example, if a CSP file uses the value of a server-side variable, *x*, as a URL parameter value, any characters within *x* can be escaped with this expression:

```
<a href="page2?ZOOM=#(..EscapeURL(x))#">Link</A>
```

If the value of *x* is `100%`, then the following text is sent to the HTTP client. The % character is escaped as `%25`.

```
<a href="page2?ZOOM=100%25">Link</A>
```

5.4.3 Escaping JavaScript with QuoteJS

The %CSP.Page class provides the **#(. .QuoteJS(x))#** string to replace all special characters with their corresponding JavaScript escape sequences.

For example, suppose a CSP file defines a client-side JavaScript function that displays a message, specified by the value of a server-side variable, *x*, in an alert box. The value of *x* is converted into a JavaScript quoted string with:

```
<script language="JavaScript">
function showMessage()
{
   alert(#(..QuoteJS(x))#);
}
</script>
```

If the value of *x* is “Don't press this button!”, then the following text is sent to the HTTP client:

```
<script language="JavaScript">
function showMessage()
{
   alert('Don\'t press this button!');
}
</script>
```

5.5 Server-Side Methods

CSP offers two techniques for invoking server-side methods from an HTML client.

- Using the HTTP Submit mechanism.
- Using the hyperevents, either **#server** (synchronous) or **#call** (asynchronous). You can also use the **HyperEventCall()** method of %CSP.Page; for details and an example, see the class reference.

The advantages of using HTTP Submit are that client-side programming is simple and no client-side components are needed. Its disadvantages are that the page is repainted by the client after a method call and server-side programming is more difficult.

If you use hyperevents, **#server** and **#call** are implemented using XMLHttpRequest. **#call** is asynchronous: if you (as a user) enter a value on a web page, the page is not updated immediately; you may have moved to another page by the time it is updated. **#server** is synchronous; the page is updated immediately on the return from the call.

Note that synchronous XMLHttpRequest is deprecated by many browsers, and in general, movement is toward only supporting the asynchronous XMLHttpRequest.

HTTP Submit and hyperevents are described further in the sections below.

5.5.1 Caché and AJAX

The acronym AJAX is commonly used to refer to a set of technologies that allows you to update a client-side page's data from the server without having to request a new browser page. Caché hyperlinks allow AJAX interactions without requiring the programmer to handle all the messy communications with the server. Two ways Caché allows AJAX transactions:

1. For CSP the use of the **#server()** and **#call()** commands allow direct calling of server-side methods from the client. (You can also use the **HyperEventCall()** method of %CSP.Page; for details and an example, see the class reference.)
2. For Zen, the programmer may define ZenMethods which handle client-server interactions. These may be synchronous or asynchronous depending on the methods signature:

Signature for a synchronized AJAX request

```
Method XYZ(arg) as %Status [ZenMethod]
```

Signature for a asynchronized AJAX request

```
Method XYZ() [ZenMethod]
```

5.5.1.1 Parallel Processing with Ajax

Ajax requests to CSP are processed sequentially on the server because of locks on nodes of `^%cspSession` global. To enable Ajax requests to be processed in parallel, if the application you are working with does not SET anything in the session global/object (so only reads), you can use the **%CSP.Session.Unlock()** method to unlock the CSP global for that session and **%session.Lock** at the end of processing.

For more information see %CSP.Session in the class reference.

5.5.2 Calling Server-side Methods via HTTP Submit

Invoking server code with an HTTP Submit requires very little functionality from the browser. It is a good technique for applications that have a wide audience and must support a wide variety of browsers. When you use an HTTP Submit, the requested page is redisplayed each time that the user clicks a *SUBMIT* button.

You can handle HTTP submits with the following:

1. Serve an HTML form containing a *SUBMIT* button:

   ```
   <form name="MyForm" action="MyPage.csp" method="GET">
   User Name: <input type="TEXT" name="USERNAME"><br>
   <input type="SUBMIT" name="BUTTON1" value="OK">
   </form>
   ```

   This defines a simple form with a text field called **USERNAME** and a *SUBMIT* button called **BUTTON1**. The *ACTION* attribute of the form specifies the URL to which the form is submitted. The *METHOD* attribute specifies which HTTP protocol is used to submit the form: `POST` or `GET`.

2. When the user clicks **BUTTON1**, the *SUBMIT* button, the browser collects the values of all controls in the form and submits them to the URL specified by the form's *ACTION* attribute. (Note that a page can submit back to itself either by specifying its name with the *ACTION* attribute or by leaving the *ACTION* attribute blank.) Regardless of whether a form is submitted via `POST` or `GET`, CSP treats the submitted values as if they were URL parameters. In this case, submitting the form is equivalent to requesting the following URL:

   ```
   MyPage.csp?USERNAME=Elvis&BUTTON1=OK
   ```

   The name and value of the *SUBMIT* button is included. If there are multiple *SUBMIT* buttons on a form, only the data button that was actually pressed is included in the request. This is the key to detecting when a **SUBMIT** has occurred.

3. The server code, in this case `MyPage.csp`, detects that a submit has occurred. It does this by testing for the name `BUTTON1` in the *%request* object:

   ```
   <script language="Cache" runat="SERVER">
      // test for a submit button
      If ($Data(%request.Data("BUTTON1",1))) {
         // this is a submit; call our method
         Do ..MyMethod($Get(%request.Data("USERNAME",1)))
      }
   </script>
   ```

4. After invoking the desired server-side logic, the server code continues and returns HTML for display by the browser. This could be a redisplay of the current form or a different page.

5.5.3 Calling Server-side Methods Using Hyperevents #server and #call

A hyperevent is our CSP extension of a web browser event and a web development technique for creating an interactive web action. Using hyperevents, you can run methods of classes on a Caché server in response to events in a client web browser without reloading the HTML page on the client. This capability is commonly called AJAX in the wider world. Caché hyperevents are useful in a number of cases, especially database applications where you may want to perform validation or search functions against a database without having to reload and reformat an entire web page. Usually, hyperevents are implemented using XMLHttpRequest.

If you are developing CSP pages using classes, you need to call the `..HyperEventHead` method during the output of the `<head>` section to load the JavaScript needed for hyperevents.

5.5.3.1 Calling Server-side Methods Using #server

Within a CSP file, you can invoke a server-side method using the **#server** directive. You can use this directive anywhere that JavaScript is allowed.

The syntax of the **#server** directive is:

```
#server(classname.methodname(args,...))#
```

where *classname* is the name of a server-side Caché class and *methodname* is the name of a method in the class. *args* is a list of client-side JavaScript arguments that are passed to the server-side method. Note that all code between the opening and closing #signs must all be on a single line.

For example, to invoke a server-side method named **Test** in the Caché class **MyPackage**, use:

```
<script language="JavaScript">
function test(value)
{
   // invoke server-side method Test
   #server(MyPackage.Test(value))#;
}
</script>
```

The CSP compiler replaces each occurrence of **#server** directive with JavaScript code that invokes the server-side method.

From a given CSP page, you can invoke a method belonging to the class generated for it using ..*MethodName* syntax. For example:

```
#server(..MyMethod(arg))#
```

5.5.3.2 Calling Server-side Methods Using #call

Synchronicity is determined by the method being called, **#server** or **#call**. **#server** is synchronous; **#call** is asynchronous.

Synchronous calls may result in a noticeable pause in the UI response (hanging). Asynchronous calls however require their own benefits and problems. For example, if the user enters a value on the web page, the page is not updated immediately; the user may have moved to another page by the time it is updated.

#server is synchronous. When you invoke a server-side method, **#server** returns a value to the client (since the client is waiting) *and* returns JavaScript for the client to execute

#call is asynchronous: When you invoke a server-side method, **#call** does not wait for a return value. Instead, your application relies on the JavaScript sent back by the server to perform any needed operations on the client.

When using asynchronous **#call**, you have to be careful when making multiple, successive calls. If you invoke a method via **#call** before the previous method is complete, the web server may decide to cancel your previous method call. Or the user might move to another page before the JavaScript executes, so the JavaScript executes on the wrong page, and result in an error. So, mostly, use **#call** when starting something running that could take a while, and provide a link to another page that shows the status of the long-running job.

As with **#server**, you can use the **#call** directive anywhere that JavaScript is allowed.

The syntax of the **#call** directive is:

```
#call(classname.methodname(args,...))#
```

where *classname* is the name of a server-side Caché class and *methodname* is the name of a method in the class. *args* is a list of client-side JavaScript arguments that are passed to the server-side method. Note that all code between the opening and closing #signs must all be on a single line.

For example, to invoke a server-side method named **Test** in the Caché class MyPackage, use:

```
<script language="JavaScript">
function test(value)
{
  // invoke server-side method Test
  #call(MyPackage.Test(value))#;
}
</script>
```

The CSP compiler replaces each occurrence of the **#call** directive with JavaScript code that invokes the server-side method.

From a given CSP page, you can invoke a method belonging to the class generated for it using ..*MethodName* syntax. For example:

```
#call(..MyMethod(arg))#
```

5.5.3.3 Hyperevent Examples

This section shows some examples of hyperevents; that is, where you use the **#server** and **#call** directives to perform server actions in response to client events. For instance: you have a form that is used to add a new customer to a database. As soon as the customer name is entered, the application checks to make sure that the customer is not already in the database. The following form definition calls a server-side **Find** method when the input contents are changed.

```
<form name="Customer" method="POST">
Customer Name:
<input type="Text" name="CName"
onChange=#server(..Find(document.Customer.CName.value))# >
</form>
```

In this case, the **Find** method could be defined in the same CSP file as:

```
<script language="Cache" method="Find" arguments="name:%String">
 // test if customer with name exists
 // use embedded SQL query

 New id,SQLCODE
 &sql(SELECT ID INTO :id FROM MyApp.Customer WHERE Name = :name)

 If (SQLCODE = 0) {
   // customer was found
   // send JavaScript back to client
   &js<alert('Customer with name: #(name)# already exists.');>
 }
</script>
```

This method communicates with the client by sending back JavaScript for execution.

Whenever a server-side method is invoked, any output it writes to the principal device is sent back to the client. There, it is converted into a JavaScript function and executed by, and in the context of, the client page.

For example, if a server-side method executes the following lines of code:

```
 Write "CSPPage.document.title = 'New Title';"
```

Then the following JavaScript is sent to the client and executed:

```
CSPPage.document.title = 'New Title';
```

In this case, this changes the title displayed in the browser to `New Title`. Any valid JavaScript can be sent back to the client in this fashion. Note that you must place a carriage return (using the ! character) at the end of each line of JavaScript or the browser cannot execute it.

To make it easier to return JavaScript from a server method, ObjectScript supports embedded JavaScript using the **&js<>** directive. This is a special language construct that lets you include lines of JavaScript in an ObjectScript method. When the method containing the embedded JavaScript is compiled, the contents of the **&js<>** directive is converted to the appropriate **Write** command statements. Embedded JavaScript can refer to ObjectScript expression using the **#()#** directive.

For example, a Caché method containing the following:

```
 Set count = 10
 &js<
   for (var i = 0; i &lt; #(count)#; i++) {
      alert('This is pleasing!');
      }
 >
```

is equivalent to:

```
 Set count = 10
 Write "for (var i = 0; i < ", count, "; i++) {",!
 Write "    alert('This is pleasing!');",!
 Write "}",!
```

When invoked from a client, this method displays a pleasing alert box 10 times.

5.5.3.4 Using #server in CSP Classes

To use hyperevents and Javascript within a CSP class, you must call hyperevent broker files explicitly. As in the example below, place **#(..HyperEventHead())#** just above the closing **<head>** tag.

```
Class esl.csptest Extends %CSP.Page [ ProcedureBlock ]
{

ClassMethod OnPage() As %Status
{

   &html<<html>
   <head>
   <script language=javascript>
   function onServer()
   {
      alert(#server(..ServerMethod())#);
   }
   </script>
   #(..HyperEventHead())#
   </head>
   <body>
   <input type=button value="click here" onclick='onServer()' />
   </body>
   </html>>
   Quit $$$OK
}

ClassMethod ServerMethod()
{
   quit "from server"
}

}
```

5.5.4 Tips for Using Server-Side Methods

The ability to invoke server-side methods from a web page is a powerful feature. There are, however, some things that you need to keep in mind when using server-side methods in an application.

Note: Within this section, anything mentioned for **#server** applies to **#call** as well unless noted otherwise.

Either the **#server** or **#call** directive can call a method on the Caché server from JavaScript in the web browser. This makes CSP able to do things such as validate a field when you move off it rather than waiting for the submission of the form and, so, give the user immediate feedback. There are several factors with using **#server** syntax that you should be aware of — otherwise it is possible to produce applications that perform very slowly, or, in some cases, do not work at all.

There are two basic rules to keep in mind when using **#server**:

1. *Never* use **#server** in the **onload** event of a web page. This can fail and it is faster and easier to generate the data when generating the web page in Caché.
2. Do not use **#server** in the **onunload** event of a web page. Use as few **#server** calls as possible and do as much work inside each call as you can because they are expensive, involving a round trip from the browser to the server.

The reason this is not a good idea is because any code that you need to run inside the **onload** event can be run faster and more easily when the page is generated from Caché. For example, suppose that you wish to setup an initial value for a JavaScript variable that can be used later in the page (maybe in **#server** calls). So you are currently doing this:

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
   var value=#server(..GetValue())#;
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>
<script language="Cache" method="GetValue" returntype="%String">
   Quit %session.Data("value")
</script>
```

However there is absolutely no need to call **#server** because the value of the JavaScript variable is already known in *%session.Data("value")* when you are generating the page. Hence, it is much better to write:

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
   var value='#(%session.Data("value"))#';
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>
```

The same trick applies no matter what you are doing, if you are updating the value of a form element when the document loads then change this to put the value in when the page is generated, for example:

```
<input type="text" name="TextBox" value='#(%request.Get("Value"))#'>
```

There is never any need to use a **#server** in the **onload** event of a page.

Because the page is unloading, it is difficult to know if the JavaScript returned from Caché will be executed or not and the actual behavior depends on the browser. Also, if the user turns off the machine, you never get an **onunload** event. Your application needs to be able to cope with this possibility in any case, probably by using timeouts on the *%session* object. You can move the **onunload #server** logic code to, for example, the start of the next CSP page that the user clicks.

5.5.4.1 Use as Few #server and #call Calls as Possible

#server and **#call** work by making the browser issue an HTTP request for a page with a special encrypted token in it that tells Caché the method name to run. Caché runs this method and any output it sends back is executed as JavaScript on the browser, in addition the **#server** call can also return a value. Because these calls both use an HTTP request, they are both roughly as expensive in network packets, CPU on the server, and so on as a normal CSP page request. If you use a lot of **#server** requests, then it dramatically reduces the scalability of your application because each **#server** call is asking for a new CSP page from the Caché server. It means that, instead of a conventional web page where you go to the URL and generate the page once, a CSP page with 10 **#server** calls in it is as costly as generating ten CSP pages; if you can reduce the number of **#server** calls you could increase the number of users the application can support by a factor of ten.

The way to reduce the number of **#server** calls is to make sure that each **#server** call you use is really needed by the application and, if it is, then make sure that this **#server** call is doing as much work as possible on the server. For example, below is a block of JavaScript that update a form with some new values from the server.

Note that this block of code uses the CSP keyword **CSPPage** to refer to the page itself, rather than the Javascript keyword **self**. In this example, the two keywords work identically. We recommend using **CSPPage** as **self** can act unexpectedly in different contexts.

```
<script language="JavaScript">
function UpdateForm()
{
   CSPPage.document.form.Name.value = #server(..workGet("Name",objid))#;
   CSPPage.document.form.Address.value = #server(..workGet("Address",objid))#;
   CSPPage.document.form.DOB.value = #server(..workGet("DOB",objid))#;
}
</script>
```

The server code is shown here. (Normally it would use an object or SQL but here we are using a global to keep the code small.)

```
<script language="Cache" method="workGet"
 arguments="type:%String,id:%String" returntype="%String">
   Quit $get(^work(id,type))
</script>
```

This single update is making three calls for new web pages from the Caché server! This can be converted to a single **#server** call that updates all the values at once, the JavaScript becomes:

```
<script language="JavaScript">
function UpdateForm()
{
   #server(..workGet(objid))#;
}
</script>
```

The method definition is:

```
<script language="Cache" method="workGet"
 arguments="id:%String" returntype="%String">
   &js<CSPPage.document.form.Name.value = #($get(^work("Name",objid)))#;
      CSPPage.document.form.Address.value = #($get(^work("Address",objid)))#;
      CSPPage.document.form.DOB.value = #($get(^work("DOB",objid)))#;>
</script>
```

So, instead of multiple calls, you just pass the data once and then make Caché do all the work. If you have a more complex JavaScript example, such as:

```
<script language="JavaScript">
function UpdateForm()
{
   CSPPage.document.form.Name.value = #server(..workGet("Name",objid))#;
   if (condition) {
      CSPPage.document.form.DOB.value = #server(..workGet("DOB",objid))#;
   }
   else {
       CSPPage.document.form.DOB.value = '';
   }
}
</script>
```

then this should still only ever need one **#server** call. You just embed the whole **if** condition into the JavaScript returned by the **#server** call, so the code **workGet** method ends up looking like:

```
<script language="Cache" method="workGet"
 arguments="id:%String" returntype="%String">
   &js<CSPPage.document.form.Name.value = #(^work("Name",objid))#;
      if (condition) {
         CSPPage.document.form.DOB.value = #(^work("DOB",objid))#;
      }
      else {
         CSPPage.document.form.DOB.value = '';
      }
   >
</script>
```

5.5.4.2 Creating Custom HyperEvent Error Handler for #server and #call

If you call something with a hyperevent (**#server** or **#call**) and, on execution, it fails to communicate with the server for some reason, then generates an error, CSP's default behavior is to display the error in an alert box. If you want to handle

the error separately, such as log it or display a different message to the user, then write a **cspRunServerMethodError** javascript function. The following example displays the error in an alert box like the default behavior:

```
function cspRunServerMethodError(errortext,error)
{
   //alert('cspRunServerMethodError - cspHyperEventErrorHandler\n\nerrortext:' + errortext +
'\n\nerror-object:\n' + JSON.stringify(error, null, 4) );

   if (error.code == '401') {
      document.location.href = '#(..Link(%request.URL))#'; //reloads the page
   }
   else {
      //...
   }

   return null;

}
```

cspHyperEventError object type has the following properties and values:

- code: corresponds to an HTTP response code or a response code from the XMLHttpRequest object in use. XMLHttpRequest codes may be browser-specific.
- text: a free text field which corresponds to the current text returned to the **cspRunServerMethodError**() callback function.
- serverCode: corresponds to the error number on the server, if available. This value may be null.
- serverText: the error message from the server, if available. This value defaults to the empty string, that is “”.
- exception: an exception which triggered the error. This value may be null.
- arguments: the list of arguments to the function where an exception was trapped. This value may be null and is only populated if exception is defined.

6

Building Database Applications

One of the most powerful aspects of CSP is that it lets you create dynamic web pages that can directly interact with a built-in, object database. This means that you can quickly build database applications that:

- Avoid the complexity of mapping relational data to objects
- Require no complex middleware
- Can be reconfigured, at runtime, from single-server to multi-tier, multi-server configurations, for true scalability

Note that by using the Caché SQL Gateway, you can build object-based CSP applications that access data in third party relational databases. Caché manages this in an application-transparent way; all the techniques described in this chapter work regardless of whether you choose to store data in the built-in Caché database or a third party database.

CSP is flexible; you can build database applications using a variety of techniques ranging from using higher-level tags that automatically bind data to HTML forms to writing server-side scripts that directly access data using objects. These techniques are outlined below.

6.1 Using Objects on a Page

Caché makes it easy to build a database of persistent objects that represent the data for an application. You can use these persistent objects in a web application in a number of ways.

The most straight-forward way to display object data on a page is to use server-side scripts to open the object and write out its contents.

The following examples use the Sample.Person class included in the Caché SAMPLES database. These examples use CSP pages but the techniques described apply to applications built by subclassing the %CSP.Page class as well.

6.1.1 Displaying Object Data in a Table

The following CSP page opens an instance of a persistent object, displays some of its properties in an HTML table, and then closes the object:

```
<html>
<body>
<script language="Cache" runat="SERVER">
 // open an instance of Sample.Person
 Set id = 1
 Set person = ##class(Sample.Person).%OpenId(1)
</script>
<table border="1">
<tr><td>Name:</td><td>#(person.Name)#</td></tr>
```

```
<tr><td>SSN:</td><td>#(person.SSN)#</td></tr>
<tr><td>City:</td><td>#(person.Home.City)#</td></tr>
<tr><td>State:</td><td>#(person.Home.State)#</td></tr>
<tr><td>Zip:</td><td>#(person.Home.Zip)#</td></tr>
</table>
<script language="Cache" runat="SERVER">
 // close the object
Set person = ""
</script>
</body>
</html>
```

If you would like to try this, copy the above code into a text file, save it as mytable.csp in your /cachesys/csp/samples directory (cachesys is the installation directory for Caché), and then point your browser at:

```
http://localhost:57772/csp/samples/mytable.csp
```

You should see data displayed in a simple HTML table.

Note: Be careful not to do any real work in the /csp/samples directory. If you upgrade Caché, it reinstalls the samples and erases your work.

6.1.2 Displaying Object Data in a Form

Using code similar to that described above, you can display data in an HTML form. This example opens an instance of a persistent object, displays some of its properties in an HTML form, and then closes the object.

```
<html>
<body>
<script language="Cache" runat="SERVER">
 // open an instance of Sample.Person
 Set id = 1
 Set person = ##class(Sample.Person).%OpenId(1)
 If ($Data(%request.Data("SAVE",1))) {
    // If "SUBMIT" is defined, then this is a submit
    // Write the posted data into the object and save it
    Set person.Name = $Get(%request.Data("Name",1))
    Set person.SSN = $Get(%request.Data("SSN",1))
    Set person.Home.City = $Get(%request.Data("City",1))
    Do person.%Save()
 }
</script>
<form method="POST">
<br>Name:
<input type="TEXT" name="Name" value="#(..EscapeHTML(person.Name))#">
<br>SSN:
<input type="TEXT" name="SSN" value="#(..EscapeHTML(person.SSN))#">
<br>City:
<input type="TEXT" name="City" value="#(..EscapeHTML(person.Home.City))#">
<br>
<input type="SUBMIT" name="SAVE" value="SAVE">
</form>
<script language="Cache" runat="SERVER">
 // close the object
Set person = ""
</script>
</body>
</html>
```

%request.Data("txt",1) is a string value if the data is less than the local variable limit in Cache. If the data is larger than this, CSP creates a stream with the value of the data in it. If long strings are disabled, the Caché variable limit is 32k. If long strings are enabled then the boundary is much bigger.

If you are creating a form that contains a field that can hold more than 32K of data, code it as follows:

```
Set value=%request.Data("fieldname",1)
If $isobject(value) {
    ; Treat this as a stream
} Else {
    ; Treat this as a regular string
}
```

6.1.3 Processing a Form Submit Request

In addition to displaying the contents of an object in a form, the preceding example also saves changes to the object when the user submits the form by clicking **Save**. This works as follows.

When a form is submitted, the values of the controls (including the button initiating the submit) are sent back to the server. In this case, the form is submitted to the same CSP page that initially served the page. You can submit to a different page by setting the value of the forms ACTION attribute.

The CSP server places the submitted values in the *%request* object Data property. The server-side script at the start of the page tests if the page is being served in response to a submit request by testing if the request parameter *Save* (which is the name of the submit button) is defined. This should only be defined as a result of a submit request. If this is a submit request, then the script copies the values submitted from the form into the appropriate properties of the object and calls the object:

```
If ($Data(%request.Data("SAVE",1))) {
   // If "SUBMIT" is defined, then this is a submit
   // Write the posted data into the object and save it
   Set person.Name = $Get(%request.Data("Name",1))
   Set person.SSN = $Get(%request.Data("SSN",1))
   Set person.Home.City = $Get(%request.Data("City",1))
   Do person.%Save()
}
```

6.1.4 <csp:object> tag

Some of the behavior in the previous examples is provided automatically by the <csp:object> tag. The `<csp:object>` tag generates the server-side code required to create or open an object instance for use on a CSP page as well as the code for closing it.

For instance, to associate a person with a page:

```
<csp:object NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<!-- Now use the object -->
Name: #(person.Name)# <br>
Home Address: #(person.Home.Street)#, #(person.Home.City)# <br>
```

In this case, the `<csp:object>` tag opens the object of class CLASSNAME with an Object ID of `1` and assigns it to the variable *person*. In an actual application, the object ID is provided from the *%request* object:

```
<csp:object NAME="person" CLASSNAME="Sample.Person"
OBJID='#($Get(%request.Data("PersonID",1)))#'>
Name: #(person.Name)# <br>
Home Address: #(person.Home.Street)#, #(person.Home.City)# <br>
```

The expression,

```
$Get(%request.Data("PersonID",1))
```

refers to the URL parameter *PersonID*.

The `<csp:object>` tag with a null *OBJID* attribute creates a new object of the specified class:

```
<csp:object NAME="person" CLASSNAME="Sample.Person" ObjID="">
```

Using the `<csp:object>` tag is equivalent to including server-side script that explicitly creates an object instance. Refer to the CSP sample page object.csp for an example using the `<csp:object>` tag.

6.2 Binding Data to Forms

CSP provides a mechanism for *binding* the data for an object to an HTML form. This binding uses the standard HTML form and input control tags to define the form allowing you to easily use any HTML editor or design tool with bound forms. The `<csp:object>` tag specifies an object instance and the attribute *cspbind* is added to the form and input control tags to indicate how they should be bound.

The CSP compiler recognizes forms containing the *cspbind* attribute and automatically generates code that:

- Displays the values of the specified object properties in the appropriate input control.
- Generates client-side JavaScript functions to perform simple validation (such as required field checking).
- Generates client-side JavaScript functions to invoke generated server-side methods for saving the bound object.
- Generates server-side methods that validate and save data input into the form. These methods can be called directly from the page using the CSP Event Broker, or you can invoke them as a result of a form submit operation.
- Generates a hidden field in the form, *OBJID*, that contains the object ID value for the bound form.

Here is a simple example of a form bound to an instance of the Sample.Person class:

```
<html>
<head>
</head>
<body>
<csp:object NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<form NAME="MyForm" cspbind="person">
<br>Name:
<input type="TEXT" name="Name" cspbind="Name" csprequired>
<br>SSN:
<input type="TEXT" name="SSN" cspbind="SSN">
<br>City:
<input type="TEXT" name="City" cspbind="Home.City">
<br>
<input type="BUTTON" name="SAVE" value="SAVE" OnClick="MyForm_save();">
</form>
</body>
</html>
```

This example uses the <csp:object> tag to open an instance of the Sample.Person class (in this case, with object ID of `1`). This object instance is named *person*. The example then binds the object instance to an HTML form by adding to its form tag an attribute called *cspbind* with the value `person`.

The form contains three text input controls, **Name**, **SSN**, and **City**, which are bound to the object properties Name, SSN, and Home.City, respectively, by adding to each of their input tags an attribute called *cspbind* whose value is the name of the object property the control is to be bound to.

Note that the names of controls used in a bound form must be valid JavaScript identifiers.

The **Name** control also has an attribute called *CSPREQUIRED*. This indicates that this is a required field (it must be given a value). The CSP compiler generates client-side JavaScript to test that the user provides a value for this field.

The last control on the form is a button that is defined to invoke the client-side JavaScript function **MyForm_save** when it is clicked. The **MyForm_save** function is automatically generated by the CSP compiler. This function gathers the values of the controls in the form and sends them to a server-side method (also automatically generated by the CSP compiler) that reopens the object instance, applies the changes made to the properties, saves the object to the database, and sends JavaScript to the client to update the values in the form to reflect what was saved.

Note that we have defined a HEAD section in this document. This is necessary when using a bound form as this is used a location for any client-side JavaScript that may be generated by the CSP compiler when it processes a bound form.

By convention, the object ID of an object used in a bound form is specified by the URL parameter *OBJID*. This makes it possible for a bound form to interact with prebuilt pages, such as those used by the CSP Search facility. To use the value of a URL parameter as an object ID, use an expression referring to it in the csp:object tag:

```
<csp:object NAME="person"
CLASSNAME="Sample.Person" OBJID=#($G(%request.Data("OBJID",1)))#>
```

6.2.1 Binding to a Property

To bind a particular HTML input control to an object property, do the following:

- Define a server-side variable that refers to an object instance using the `csp:object` tag.
- Create an HTML form using the form tag. Bind the form to the object instance by adding a *cspbind* attribute to the form tag. Make the value of the *cspbind* attribute the name of the `csp:object` tag.
- Create an HTML input control in the form and add a *cspbind* attribute to it. Make the value of this *cspbind* attribute the name of the object property to bind to.

The *cspbind* attribute lets you bind to many different types of object properties. This is detailed in the following table:

Table 6–1: Effects of cspbind Attribute

Property	Example	Effect
Literal	cspbind=“Name”	Bind the control to a literal property. Display the *DISPLAY* value of the property.
Property of Embedded Object	cspbind=“Home.City”	Bind the control to a embedded object property. Display the *DISPLAY* value of the embedded object property.
Referenced Object	cspbind=“Company”	Bind the control to the object ID value for a reference property. Display the *object ID* value for the reference property.
Property of Referenced Object	cspbind=“Company.Name”	Bind the control to a property of a referenced object. Display the *DISPLAY* value of the referenced object property.
Instance Method	cspbind=“%Id()”	Bind the control to return value of an instance method. Display the return value of the method as a read-only field.

The binding mechanism can be used with most of the available HTML input controls. This is detailed in the following table:

Table 6–2: HTML Input Elements Supported by cspbind

Control	Effect
input type="TEXT"	Display the value of a property in a text control.
input type="PASSWORD"	Display the value of a property in a password control.
input type="CHECKBOX"	Display the value (as a boolean) of a property in a check box control.
input type="RADIO"	Display the value of a property by selecting the radio button whose value corresponds with the property value.
input type="HIDDEN"	Display the value of a property in a hidden control.
SELECT	Display the value of a property by selecting the choice in the SELECT list whose value corresponds with the property value. You can populate the choices in the SELECT list using a class query by also specifying *CLASSNAME*, *QUERY*, and optional *FIELD* attributes. Refer to the CSP sample page form.csp for an example.
IMAGE	Display a binary stream property in an `IMAGE` tag.
TEXTAREA	Display a property value as text in a `TEXTAREA` control.

6.3 CSP Search Page with <csp:search> Tag

The csp:search tag creates a generic `search` page that you can use in conjunction with bound forms to perform lookup operations.

An application user can access the CSP Search page from a page containing a bound form and use it to find objects in the database that match a set of criteria. The user can then select one of these objects and edit it.

The `csp:search` tag generates a client-side JavaScript function that displays a search page. The search page is displayed by the %CSP.PageLookup class.

The `csp:search` tag includes attributes that give you control over the operation of the search page. These include:

Table 6–3: <csp:search> Tag Attributes

Attribute	Description
CAPTION	Optional. A caption string displayed in the standard search page.
CLASSNAME	Required. The name of the class upon which the search is performed.
FEATURES	Optional. A string contains the *features* argument passed to the JavaScript **window.open** method when a popup search window is used. This gives you greater control over how popup windows are displayed.
MAXROWS	Optional. Specifies the maximum number of rows to display in the search results table. The default is 100.
NAME	Required. The name of the generated client-side JavaScript function that invokes the search page.

Attribute	Description
OBJID	The Object ID value of the object displayed when the search page was invoked. This is used to redisplay the old page when the user cancels a search.
ONSELECT	Optional. In a popup search page, the name of a JavaScript function to call when the user selects a specific search result. This function is called with the Object ID value of the selected object.
OPTIONS	Optional. A comma-delimited list of search page options. These options include "popup" to create a popup search window and "predicates" to display a drop down list of search predicates.
ORDER	Optional. A name of a field to order the search results on.
SELECT	Optional. A comma-delimited list of fields to display in the search result table. If not specified, the WHERE list is used as the SELECT list.
STARTVALUES	Optional. A comma-delimited list of the names of controls in the form invoking the search page whose contents are used as *seed* values in the search page. The order of names in the list corresponds to the criteria fields (specified by the WHERE attribute) in the search page.
TARGET	Optional. In a non-popup search page, specifies the name of the page to which the links in the search result window point. That is the page to display when the user makes a selection. The default is the page invoking the search.
WHERE	Required. A comma-delimited list of fields used as criteria for the search page. These fields are also shown in the search result table unless the SELECT attribute is specified.

For example, the following defines a JavaScript function, **MySearch**; this function displays a popup search window that searches for Sample.Person objects by name:

```
<csp:search NAME="MySearch" WHERE="Name" CLASSNAME="Sample.Person"
 OPTIONS="popup" STARTVALUES="Name" ONSELECT="MySearchSelect">
```

The **ONSELECT** callback function for this search page looks like this.

```
<script language="JavaScript">
function MySearchSelect(id)
{
    #server(..MyFormLoad(id))#;
    return true;
}
</script>
```

This function uses the CSP **#server()#** directive to invoke the server-side method **MyFormLoad**. The **MyFormLoad** method is automatically generated as a result of binding the HTML form `MyForm` to an object using *cspbind*. This method populates the contents of the form with the property values of an object with object ID *id*.

For additional examples, refer to the CSP sample pages form.csp and popform.csp.

6.4 Enabling Logging in ISCLOG

To troubleshoot CSP issues, enable logging for Caché by entering the following command in the Terminal:

```
 Set ^%ISCLOG = 2
```

You can view logging information in the ^ISCLOG global. This global logs events in Caché for use in debugging. For reference, the log levels are as follows:

- 0 — Caché performs no logging.
- 1 — Caché logs only exceptional events (such as error messages).
- 2 — Caché logs detailed information, such as `method ABC invoked with parameters X,Y,Z and returned 1234`.
- 3 — Caché logs raw information such as data received from an HTTP request.

You can turn Caché logging off with either

```
Set ^%ISCLOG = 0
```

or

```
Kill ^%ISCLOG
```

In ISCLOG, some entries match Event Log header fields as follows:

ISCLOG	Event Log
Job	Cache-PID
SessionId	Session-ID
Tag	Request-ID

Fields and definitions in ISCLOG are shown in the table below.

Table 6–4: ISCLOG Fields

Field	Definition
%category	`CSPServer`: Logged from `cspServer`, `cspServer2`, `%request`, `%response`.
	`CSPSession`Logged from `%session` and parts of `cspServer` and `cspServer2` which handle a session. This allows watching the lifecycle of a session.
	`CSPLicense`Logged from parts of `cspServer` and `cspServer2` which handle a licensing.
	`Gateway Request`Logged from the `GatewayMgr`, `GatewayRegistry`, the Gateway request handler and parts of `cspServer2` which handle gateway requests.
%level	1= Exceptions and errors.
	2=`CSPSession` information. `CSPLicense` information. Information from `cspServer`: the part of the request handling after the `%response`, `%session`, and `%request` have been setup. This includes authentication, license handling, redirection, and calling the CSPpage.
	3=Information from `cspServer2`: the part of handling the request which sets up the `%response`, `%session`, `%request`, and hand-shaking/data transfer with the CSP Gateway.
%job	The value of `$job` when the ISCLOG request was made. Matches the Cache-PID field from the Event Log header.

Field	Definition
%sessionid	Entered when available. The value of `sessionid` at the time the ISCLOG request was made. Matches the Session-ID field from the Event Log header.

%tag	For the CSP Server, the tag contains the Request id from the gateway (when available). This matches the `Request-ID` field from the Event Log header. Other loggers may set this value to any value.
	Available for use by creators of ISCLOG entries. Stores ID of the request sent to it by the CSP Gateway. It can be used as a filter for generation of ISCLOG entries.
	`Set ^%ISCLOG("Tag","mytagvalue1")=1`
	`Set ^%ISCLOG("Tag","mytagvalue2")=1`
	Only ISCLOG requests with no tag or with tags of "mytagvalue1" or "mytagvalue2" will be recorded.

%routine	The name of the routine currently being executed.
%message	See the sectionMessage Format below.

6.4.1 Message Format

Messages start with the name of the tag label or method currently being executed. This name is enclosed in square brackets. `[MyMethod] rest of messages`.

Messages in the `CSPSession` category also have `CSPSession-Id=sessid` after the method name. This is needed as session events can be logged before the session is created or after it was destroyed, meaning the SessionId field is empty in the ISCLOG entry.

```
[MyMethod] CSPSession-Id: 12ty34ui22
```

Messages in the `GatewayRegistry` category also have `CSPID=cspid`(when available) after the method name. This allows the tracking of an individual gateway request from the API call through the Gateway Request Handler.

```
[MyMethod]CSPID:334r43345 rest of message
```

7

Localizing Text in a CSP Application

This chapter describes how to localize text in a CSP application in the common scenario where you are using class-based development. For tag-based development, see "Localization and Tag-Based Development," later in this book.

Also see the article *String Localization and Message Dictionaries*.

7.1 Localization Basics

When you *localize* the text for an application, you create an inventory of text strings in one language, then establish a convention for substituting translated versions of these messages in another language when the application locale is different.

Caché supports the following process for localizing strings:

1. Developers include localizable strings within their code. Within a CSP application, the easiest approach is to use class-based development and to use one of the `$$$Text` macros. In the place of a hardcoded literal string, include an instance of the `$$$Text` macro (or a related macro), providing values for the macro arguments as follows:

   - The default string
   - The domain to which this string belongs (localization is easier to manage when the strings are grouped into domains)
   - The language code of the default string

   For example, instead of this:

   ```
   &html<<div>"Hello world"</div>>
   ```

   Include this:

   ```
   set hello=$$$Text("Hello world","sampledomain","en-us")
   &html<<div>#(hello)#</div>>
   ```

2. When the code is compiled, the compiler generates entries in the message dictionary for each unique instance of the `$$$Text` macro (and its related macros).

   The message dictionary is a global and so can be easily viewed (for example) in the Management Portal. Caché provides class methods to help with common tasks.

3. When development is complete, release engineers export the message dictionary for that domain or for all domains.

   The result is one or more XML message files that contain the text strings in the original language.

4. Release engineers send these files to translators, requesting translated versions.

5. Release engineers import the translated XML message files into the same namespace from which the original was exported.

   Translated and original texts coexist in the message dictionary.

6. At runtime, the application chooses which text to display based on the browser default language.

For steps 1 and 2, also see the appendix "Localization and Tag-Based Development."

For information on exporting and importing the message dictionary, see the article *String Localization and Message Dictionaries*.

7.2 $$$Text Macros

Caché provides three related `$$$Text` macros (in %occMessages.inc, which is included in %occInclude.inc):

- `$$$Text` returns a %String
- `$$$TextJS` returns a %String that is escaped for use in JavaScript
- `$$$TextHTML` returns a %String that is escaped for use in HTML

Each of these macros takes three arguments: the default string, the domain to which this string belongs, and the language code of the default string. When code is compiled, the compiler generates entries in the message dictionary for each unique set of values of the arguments.

The %String returned by `$$$Text` may be assigned to a variable, which you can use to represent the message in subsequent calls. For example:

```
Set tmsg = $$$TextJS("Error saving production")
&js<alert('#(tmsg)#: #($ZCVT($ZE,"O","JS"))#');>
```

Or, you can simply insert a `$$$Text` macro anywhere you need a string:

```
&js<alert('#($$$TextJS("Error saving production"))#: #($ZCVT($ZE,"O","JS"))#');>
```

7.2.1 Argument Details

Formally, the `$$$Text`, `$$$TextJS`, and `$$$TextHTML` macros take the following arguments in order:

Argument	Description
text	Non-empty string. *text* must be a literal string. It cannot be the value of a CSP runtime expression enclosed in #()# syntax. The format used for *text* may be: "*actualText*" Or: "*@textId@actualText*" Where *textId* is a message ID and *actualText* is the text of the message. The string *actualText* may consist of any of the following items, separately or in combination: • Simple text, as permitted by the file format • Substitution arguments *%1*, *%2*, *%3*, or *%4* • HTML formatting • An ObjectScript string expression If provided, the *textId* is used as the message ID. If *@textId@* is not specified, the system generates a new *textId* by calculating the 32–bit CRC (Cyclic Redundancy Check) of this text. If the *textId* is specified and a message already exists with this ID, the existing message is checked to see if it has the same text as *actualText*. If not, an error is reported.
domain	(Optional) String specifying the domain for the new message. If not specified, *domain* defaults to the value of the *DOMAIN* class parameter at compile time and *%response.Domain* at runtime.
language	(Optional) RFC1766 code specifying the language. Caché converts this string to all-lower-case. If not specified, *language* defaults as follows: • At compile time: `$$$DefaultLanguage`. • At runtime: *%response.Language*, or if no value is defined for *%response.Language* then `$$$DefaultLanguage`. Tag-based CSP pages automatically acquire a value for *%response.Language* from browser settings, so it is available as a default language. This is not true for class-based CSP pages, which must explicitly set a value for *%response.Language* to use it as a default. You can assign a value to *%response.Language* by giving it the return value of the %Library.MessageDictionary class method MatchLanguage(), discussed later in this chapter. Given a list of languages and a domain name, this method uses HTTP 1.1 matching rules (RFC2616) to find the best-match language within the domain.

7.2.2 $$$Text at Compile Time

When you compile a class that contains calls to `$$$Text`, `$$$TextJS`, or `$$$TextHTML` macros, each call generates a message in the message dictionary, with *text*, *message ID*, *domain*, and *language* as provided by the macro arguments.

The first time a message is added to a domain by `$$$Text`, `$$$SessionLanguage` is used whether the language argument is specified or not. Subsequent `$$$Text` macros for the same domain add messages with the same language as the first added message.

7.2.3 $$$Text at Runtime

If the message text contains arguments (*%1*, *%2*, *%3*, *%4*) you must specify the corresponding substitution text before displaying the text. Because $$$Text returns a string, you can use any string operation native to your coding language. For example, in JavaScript:

```
var prompt = '#($$$TextHTML("Remove user %1 from this Role?"))#';
prompt = prompt.replace(/%1/g,member.userName);
```

You can also use the $$$Text string as the first argument of the %response.FormatText method or a $$$FormatText macro.

7.3 Other Options for Displaying Localized Strings

The easiest way to display a localized string at runtime is to use one of the `$$$Text` macros as described earlier in this chapter.

This topic explains other ways to retrieve message text from the message dictionary at runtime. If the message text contains arguments (*%1*, *%2*, *%3*, *%4*) you must also specify the corresponding substitution text before displaying the text on the page.

7.3.1 %response.GetText Method

The %CSP.Response class offers a **GetText** instance method that enables you to retrieve text from the message dictionary and substitute values for any arguments the message may have. In CSP class code, the currently instantiated %CSP.Response object is represented by the variable *%response*. This topic refers to the method as **%response.GetText**.

The method signature is:

```
method GetText(language As %String = "",
               domain As %String = "",
               id As %String,
               default As %String,
               args...) returns %String
```

Argument	Description
domain	(Optional) A string specifying the domain for the message. If not specified, *domain* defaults to `%response.Domain`.
language	(Optional) An RFC1766 code specifying the language. Caché converts this string to all-lowercase. If not specified, *language* defaults to the value of %response.Language, which automatically takes its runtime value from the browser settings.
id	The message ID.
default	The string to use if the message identified by *language*, *domain*, and *id* is not found.
arg1, *arg2*, and so on	Substitution text for the message arguments. All of these are optional, so you can use **%response.GetText()** even if the message has no arguments.

7.3.2 FormatText Method

The %Library.MessageDictionary class offers a **FormatText()** class method that enables you to substitute text for message arguments. You can use **FormatText()** when you already have the message text from the message dictionary.

The method signature is:

```
ClassMethod FormatText(text As %String, args...) As %String
```

Argument	Description
text	The message text. Use a %String returned by %response.GetText or $$$Text.
arg1, *arg2*, and so on	Substitution text for the message arguments.

7.3.3 $$$FormatText Macros

These macros enable you to substitute text for message arguments. You can use them when you already have the message text from the message dictionary:

- `$$$FormatText`
- `$$$FormatTextJS` (applies JavaScript escaping to the $$$FormatText result)
- `$$$FormatTextHTML` (applies HTML escaping to the $$$FormatText result)

These macros are in %occMessages.inc, which is included in %occInclude.inc.

The `$$$FormatText` macro returns a %String. The syntax is:

`$$$FormatText(`*text*`,`*arg1*`,`*arg2*`,...)`

Argument	Description
text	The message text. Use a %String returned by %response.GetText or $$$Text.
arg1, *arg2*, and so on	Substitution text for the message arguments.

7.4 The MatchLanguage() Method

You may need to set the *Language* property of the CSP response. To do so, set the *%response.Language* property, using the value returned by the **MatchLanguage()** method of %MessageDictionary:

```
Set language = ##class(%MessageDictionary).MatchLanguage(languages,domain,flag)
```

This finds the best language match to a language in the list of languages for the specified domain. The method uses HTTP 1.1 matching rules (RFC2616). The list of languages is a comma-separated list of RFC1766 format language names. Each language in the list *may* be given an associated quality value which represents an estimate of the user's preference for the languages specified by the list of languages. The quality value defaults to `q=1`.

For example, `da, en-gb;q=0.8, en;q=0.7` would mean: `I prefer Danish, but will accept British English and other types of English.` A language from the list matches a supported language tag if it exactly equals the tag, or if it exactly equals a prefix of the tag such that the first tag character following the prefix is a hyphen (-). The special language asterisk (*), if present in the input list, matches every supported language not matched by any other language present in the list.

The language quality factor assigned to a supported language tag is the quality value of the longest language in the list that matches the language-tag. The language that is returned is the supported language that has been assigned the highest quality factor.

The s flag (system) is an optional flag indicating whether system or application messages are to be matched.

8

Developing Custom Tags

This chapter covers the development and use of custom tags. Its topics include:

- Rules and Actions
- Tag Matching
- Server-Side Expressions and Code in Rule Actions
- Server Document Object Model
- Using <csr> Tags in Actions
- Using <csr> Additional Tags Outside Actions
- Using <csr> Rule Classes
- Using <csr> %CSP.Rule Methods
- Using <csr> %CSP.AbstractAtom Methods
- Using <csr> Available Routines
- Using <csr> Example

CSP allows you to develop custom HTML tags for use in CSP files. The CSP markup language is itself implemented using the custom tag mechanism. Custom tags provide a way:

- To provide HTML authors with additional functionality using familiar syntax
- To develop reusable components for web applications

8.1 Rules and Actions

The CSP compiler converts marked-up HTML documents into Caché classes (see "CSP compiler"). Some of the power of the CSP compiler comes from its ability to

1. Recognize **#( )#** expressions embedded in a CSP document, and
2. Recognize certain tags, or combinations of tags, in an HTML (or XML) document and replace them with developer-supplied actions. This is called *tag matching* and is described in this chapter.

If you are familiar with XML's XSL technology, you may recognize that CSP is a variant of XSL with additional features added to generate HTML for applications.

The following CSP page example contains a custom tag, <my:COMPANY>, that displays your company name in an HTML page:

```
<html>
<body>
<my:COMPANY>
</body>
</html>
```

When this page is processed, we want the CSP compiler to replace the <my:COMPANY> tag with suitable HTML text, for instance:

```
<html>
<body>
<b>Octoglomerate</b>
</body>
</html>
```

The action for the CSP compiler to take for the <my:COMPANY> tag is defined in a *rule file* with a <csr:action> tag. A rule file is an XML document with a .csr (Caché Server Rule) file extension that defines rules for recognizing tags and the actions to be performed when those tags are found. It also can include additional information, such as a description of the rule.

The rule file for the<my:COMPANY> tag might look like this and might be named company.csr:

```
<csr:rule name="myCOMPANY" match="my:COMPANY" empty>
<csr:action>
<b>Octoglomerate</b>
</csr:action>
</csr:rule>
```

This rule file specifies this:

Using the <csr:rule> tag, the file defines a rule named myCOMPANY. The attribute empty specifies that the <my:COMPANY> tag has no closing tag.

A rule name has the same naming restrictions as Caché classes. (For information on Caché naming rules, see "Naming Conventions" in *Using Caché Objects* or "Syntax Rules" in *Using Caché ObjectScript*). A rule is active only in the Caché namespace in which it is defined, except for rules starting with %, which are defined in the %SYS namespace and are visible by all other namespaces.

The <csr:action> tag specifies the action to take. The contents of the <csr:action> tag must be HTML and any of the following:

- **#()#** expression
- **##()##** expression
- <script> tag
- Additional <csr:> tags (these are discussed later).
- The <csr:action> tag cannot contain CSP tags; that is, the CSP compiler does not perform tag transformations on the contents of an action.

To load the company.csr rule file, move the file to the /cachesys/csp/user directory, start a Caché session (using the Terminal), and execute the following command:

```
Do $System.CSP.LoadRuleFile("company.csr")
```

This loads the rule definitions in the specified file into the current namespace. You can also use Studio to load and compile rule files with **File > New > Caché Server Page**. Save the file in the /csp/user directory. You can observe which rules are in effect using the rulemgr.csp page provided with the CSP samples.

Much of CSP is implemented using rule files. You can view the system rules files in Studio, %SYS namespace, **Workspace** window, **Namespace** tab, under **CSP files**.

8.2 Tag Matching — match Attribute

Specify attribute values for a tag by placing them within square brackets, `[ ]` following a tag name.

The *match* attribute of the `<csr:rule>` tag defines what the CSP compiler should recognize as a rule and then perform the specified action. The *match* attribute is a string of one or more tag names separated by the `/` (slash) character. If there is more than a single tag name, they are assumed to be nested, specified left to right, from the outermost to the innermost tag. The `*` (asterisk) character is a wildcard that matches any tag.

To better illustrate this, let us look at some examples of *match* values in the table below.

A rule is fired for the inner-most tag of the match. If there are multiple rule definitions for the same tags, the CSP compiler determines which rule to use by determining which *match* value is most specific. For example, a rule for `AAA/ BBB` is more specific than a rule for `BBB`. Similarly a rule that specifies attribute values, such as `BBB[CCC]`, is more specific than one that does not, `BBB`.

Table 8–1: Examples of Tag Matching

Value of match	Rule is fired:
AAA	Whenever an `<AAA>` tag is encountered: `<AAA></AAA>`
AAA/BBB	Whenever a `<BBB>` tag that is directly nested in an `<AAA>` tag is encountered: `<AAA> <BBB></BBB> </AAA>`
AAA/*/BBB	When a `<BBB>` tag is encountered nested anywhere in an `<AAA>` tag: `<AAA><FFF> <BBB></BBB> </FFF></AAA>`
AAA[CCC]	When an `<AAA>` tag with a *CCC* attribute (having any value) is encountered: `<AAA CCC=”10” ></AAA>`
AAA[CCC=22]	When an `<AAA>` tag with a CCC attribute having a value of “22” is encountered: `<AAA CCC=”22”><AAA>`
AAA[CCC=22]/*/BBB	When a `<BBB>` tag is encountered nested anywhere in an `<AAA>` tag that has a *CCC* attribute with a value of “22” : `<AAA CCC=”22” ><FFF> <BBB></BBB> </FFF></AAA>`

8.3 Server-side Expressions and Code in Rule Actions

Actions in rules may contain expressions and code that are executed either when the page is executed (runtime) or when the page is being compiled (compile-time).

8.3.1 Runtime Expressions in Actions

To specify a runtime expression in an action, use the same **#(expr)#** syntax that can be used in a CSP page. For example, here is the definition of a rule that defines a `<TODAY>` tag that displays the current time using the Caché **$ZDATE** command:

```
<csr:rule name="TODAY" match="TODAY" empty>
<csr:action>
Today is: <b>#($ZDATE($H))#</b>
</csr:action>
</csr:rule>
```

If you load this rule, you can place it in the body of a CSP page:

```
<TODAY>
```

And when the page is requested, the current date is displayed.

8.3.2 Compile-time Expressions in Actions

To specify a *compile-time* expression in an action, use the **##(expr)##** syntax. For example, here is the definition of a rule that defines a tag, `<LASTMOD>`, that displays the time that a CSP page was last compiled. The **##( )##** expression is evaluated when the page is compiled. The results of evaluating the expression become a static part of the generated CSP page.

```
<csr:rule name="LASTMOD" match="LASTMOD" empty>
<csr:action>
This page was last compiled on: <b>##($ZDATE($H))##</b>
</csr:action>
</csr:rule>
```

You can include compile time expressions in runtime expressions. In the case below, the first **$H** is evaluated at runtime, giving the current date. The second is evaluated when the page is compiled returning the date on which the page was compiled.

```
This page is #(+$H - ##(+$H)##)# days old.
```

8.3.3 <script> Tags in Actions

Similarly, you can include multiple lines of code in an action using the `<script language=cache runat=server>` tag for runtime code and the `<script language=cache runat=compiler>` tag for compile-time code. For example, here is a rule that creates an unordered list with 100 items in it:

```
<csr:rule name="BIGLIST" match="BIGLIST" empty>
<csr:action>
<ul>
<script language="Cache" runat=server>
For i = 1:1:100 {
    Write "<li>Item " _ i _ $C(13,10)
}
</script>
</ul>
</csr:action>
</csr:rule>
```

If you load this rule, you can place it in the body of a CSP page like this:

```
<BIGLIST>
```

And when the page is requested, an unordered list with 100 items is displayed.

8.4 Server Document Object Model

When the CSP compiler processes a CSP document, it first looks for all tags that are involved in rule matching. As the compiler scans the document it creates a tree of objects that match the structure of the tags contained in the CSP document. This tree is referred to as the *server-side document object model* and is directly analogous to the document object model available in a browser when an HTML page is displayed.

The server-side document object model consists of instances of subclasses of the %CSP.AbstractAtom class, representing a unit of an HTML document. An HTML document consists of two types of atom objects, *Rule* and *TextAtom*, each represented by their respective class: %CSP.Rule and %CSP.TextAtom. An Element, represented by an instance of a subclass of %CSP.Rule, represents an HTML tag, a collection of its attribute values, its inner HTML, and a collection of any inner tags it may contain. A `TextAtom` represents anything that is not an Element. For efficiency, the CSP compiler only creates *Element* objects for tags that are involved in rule matching; the rest (for example, `<B>` and `<I>` tags) are contained in *TextAtom* objects.

For example, assuming there is a rule for the `<MYTAG>` tag, the following CSP document:

```
<html>
<body>
Hello!
<MYTAG MSG="Welcome">
</body>
</html>
```

constructs the following server-side document objects:

- An *Element* with TagName of `HTML` containing the children:
  - An *Element* with TagName of `body` containing the children:
    - A *TextAtom* with Text of `Hello!`.
    - An *Element* with TagName of `MYTAG` and attribute (*MSG*) of `Welcome`.

The server-side document object model is only created during page compilation; it does not exist at runtime (again for efficiency). This is the main reason why actions may contain expressions and code that are executed at compile time: to take advantage of the server-side document object model.

After the CSP compiler constructs the document object model, it processes the document by visiting the objects in the tree, depth first, and firing the rules associated with each %CSP.Rule object in the tree and rendering the results into executable code. %CSP.TextAtom objects, by definition, do not have rules, and thus they are rendered directly into executable code.

8.4.1 Access Rule Attribute Values

When a rule is executed, a reference to the %CSP.Rule object associated with it is available via the current object.

For example, suppose you want to define a custom `<MESSAGE>` tag that displays the message specified by its *VALUE* attribute using bold, italicized text:

```
<MESSAGE VALUE="Yo!">
```

A rule definition for this might look like:

```
<csr:rule name="MESSAGE" match="MESSAGE" empty>
<csr:action>
<B><I>##(..GetAttribute("VALUE"))##</I></B>
</csr:action>
</csr:rule>
```

The `MESSAGE` rule is fired (that is, **RenderStartTag** is called) whenever the `<MESSAGE>` tag is encountered. Its actions are to:

1. Write out a `<B>` and an `<I>` tag.
2. Write out the value of the tag object
3. Close the `<B>` and `<I>` tags.

8.5 Using <csr> Tags in Actions

Within the action definition of a rule, there are some additional tags that can be used. This section describes:

- <csr:default> Tag
- <csr:children> Tag
- <csr:section> Tag

8.5.1 <csr:default> Tag

The `<csr:default>` tag directly renders the contents of the tag associated with this rule. For example, the following rule writes out the `<ECHO>` tag and any attributes and/or children it may have:

```
<csr:rule name="ECHO" match="ECHO" >
<csr:action>
<csr:default>
</csr:action>
</csr:rule>
```

This tag is mainly used for cases where you want to change some aspect of a tag but not otherwise disturb it. For example, if you want all tables on your CSP pages to have a red background, define a rule for the `<table>` tag:

```
<csr:rule name="REDTABLE" match="TABLE" >
<csr:action>
<script language="Cache" runat="COMPILER">
    // set the bgcolor attribute for this element
    Do ##this.SetAttribute("BGCOLOR","red")
</script>
<csr:default>
</csr:action>
</csr:rule>
```

When this rule is fired, it changes the value of the `BGCOLOR` attribute for any `<TABLE>` tag to `red` (using a compile-time script) and then render the table tag (and its children) unaltered in every other respect.

8.5.2 <csr:children> Tag

The `<csr:children>` tag writes out any child tags that a tag may have. It differs from the `<csr:default>` tag in that it does not render the tag associated with this rule. Use this tag when you want to have complete control over how an enclosing tag is rendered but do not want to worry about how the children are rendered.

8.5.3 <csr:section> Tag

The `<csr:section>` tag specifies a specific location in the resulting HTML page where its contents are to be rendered. By default, an action writes out text at the location in the runtime HTML page equivalent to the location of the rule tag in the CSP document. The `<csr:section>` tag allows you to change this. For example: you want to define a rule that creates a button in the body of an HTML page and some corresponding JavaScript in the head section of the page. You could do this with this rule:

```
<csr:rule name="MYBUTTON" match="FORM/*/MYBUTTON" empty>
<csr:action>
<csr:section NAME=HEAD>
<script language="JavaScript">
function MyButton()
{
 alert('MyButton pressed!');
 return true;
}
</script>
</csr:section>

<input type="button" value='##(##this.GetAttribute("VALUE"))##'
onclick="MyButton();"></input>
</csr:action>
</csr:rule>
```

8.6 Using <csr> Tags Outside Actions

There are <csr> tags that can be used outside an actions in the rule definition. This section describes the following tags:

- <csr:class> Tag
- <csr:property> Tag
- <csr:description> Tag
- <csr:attribute> Tag

8.6.1 <csr:class> Tag

The <csr:class> tag allows the use of IMPORT, SUPER or INCLUDES attributes, enabling the generated rule class to have access to other class methods.

An example of this can be seen in the rule definition for the <csp:else> tag, where %CSP.RuleBlock is specified as the superclass (by default, %CSP.RuleElement is the superclass of all classes that represent elements in the DOM (Document Object Model) model for a CSR page).

```
<csr:rule name="%ELSE" match="CSP:IF/CSP:ELSE" empty>
<csr:class super=%CSP.RuleBlock>
<csr:action>
<SCRIPT LANGUAGE=Cache RUNAT=COMPILER>
    New ifblock
    Set ifblock=..GetCurrentBlock()
    If ifblock'="" {
        If ifblock.EndLabel="" Set ifblock.EndLabel=..GetNewLabel()
        Do ..WriteServer(" Goto "_ifblock.EndLabel_" ;}")
        Do ..WriteServer(ifblock.NextLabel_" ;{")
        Set ifblock.NextLabel=""
    }
</SCRIPT>
</csr:action>
</csr:rule>
```

8.6.2 <csr:property> Tag

The <csr:property> tag defines a property in the rule class. This permits you to specify a property for the generated csr class. One use of <csr:property> is to set a property during the rendering of the start tag, and then check that same property during the rendering of the end tag. The *name* attribute specifies the name of the property. The following attributes of the property are supported:

- *name*
- *description*

- *final*
- *initial*
- *multidimensional*
- *private*
- *transient*
- *type*

8.6.3 <csr:description> Tag

The `<csr:description>` tag contains the description and, optionally, examples, of the custom rule.

The following is an example of the tag taken from the %SQLCURSOR rule:

```
<csr:description>
    The <b>SCRIPT LANGUAGE=SQL</b> creates embedded SQL for a
    DECLARE CURSOR statement in the class generated by the CSP page.  The
    declared cursor will be opened using an SQL OPEN statement and the
    SQLCODE will be returned.  It is the programmers responsibility to
    display any error indicated by the SQLCODE value.<p>
    The mode of the query is determined by the MODE attribute.  The mode is
    taken to be a runtime mode by default.  If the COMPILERMODE attribute
    is specified, then the mode is taken to be a compile time mode that
    is defined by a generated #SQLCOMPILE statement.
    <p>For example:
    <EXAMPLE>
    <SCRIPT LANGUAGE=SQL CURSOR=prsn>
      select name,ssn from sample.person where ssn %STARTSWITH '2' order by ssn
    </script>
    <CSP:WHILE CURSOR=prsn INTO='Name,ssn' counter=x condition=(x<3)>
      Name: #(Name)#<br>
      SSN: #(ssn)#<br>
    </CSP:WHILE>
    </EXAMPLE>
    <p>Will display all people whose ssn begins with 1:
    <OUTPUT>
    Name: Smith, Joe<br>
     SSN: 111-11-1111<br>
    Name: Jones, Harry<br>
     SSN: 111-11-1122<br>
    and so on..<br>
    and so on..<br>
    </OUTPUT>
</csr:description>
```

8.6.4 <csr:attribute> Tag

The `<csr:attribute>` tag is used to contain a list of a custom tag's attributes, along with a brief description of each.

The following example is taken from the `<csp:content>` tag:

```
<csr:attribute
        name=Type
        description="Specify the default Content-Type"
        type="contentType:STRING"
        >
<csr:attribute
        name=Charset
        description="Specifies the default charset"
        type="charSet:STRING"
        >
<csr:attribute
        name=NoCharSetConvert
        description="Turns off the charset conversion"
        type="noCharSetConvert:STRING"
        >
```

8.7 Using Rule Classes

The Rule Compiler generates a class for each rule definition that is compiled. It is this code that is executed when a rule is matched. This means that

1. Rules can be more powerful
2. You can create rules directly as classes, and
3. You can view and edit rule classes in Studio.

8.7.1 Structure of Generated Rule Classes

When a rule is compiled from a .csr file, the rule class that is created consists of a **RenderStartTag** method and, if compile-time code was specified in the rule definition, one or more **CompilerMethod** methods. A **RenderEndTag** method is also added to the class if either the `<csr:children>` or `<csr:default>` tag is in the rule definition. While the **CompilerMethod** methods contain code to be executed at compile time, the **RenderStartTag** and **RenderEndTag** methods each consist of a series of **Write** methods for code expressions to be written directly to the CSP page class. The type of **Write** method used depends on the expression. The **Write** methods are defined in the %CSP.AbstractAtom class, and are covered later in this chapter.

Below is a rule class belonging to the `<csr:if>` rule. It contains a **RenderStartTag** and **RenderEndTag** method, and has two instances of **CompilerMethod**. Each of these methods is explained in further detail below.

```
Import User

Class %csr.csp.IF Extends %CSP.RuleBlock
{

Parameter CSRURL = "w:/csp/rules/Control.csr";

Method CompilerMethod1() [ Language = cache ]
{
 Do ..NewBlock()
 Set ..NextLabel=..GetNewLabel()
 Do ..WriteServer(
   " If '("
     _ $$UnEscapeHTML^%cspQuote(..GetAttribute("condition","0"))
     _ ") Goto "
     _ ..NextLabel
     _" ;{"
   )
}

Method CompilerMethod2() [ Language = cache ]
{
 New comment Set comment=" ;}"
 If ..EndLabel'="" Do ..WriteServer(..EndLabel_comment) Set comment=""
 If ..NextLabel'="" Do ..WriteServer(..NextLabel_comment)
 Do ..RemoveBlock()
}

Method RenderEndTag() As %Status
{
 New element Set element=##this
 Set %statuscode=$$$OK Do ..CompilerMethod2()
 Quit:$$$ISERR(%statuscode) %statuscode
 Quit $$$OK
}

Method RenderStartTag() As %Status
{
 New element Set element=##this
 Set %statuscode=$$$OK Do ..CompilerMethod1()
 Quit:$$$ISERR(%statuscode) %statuscode
 Quit $$$PROCESSCHILDREN
}

}
```

8.7.2 RenderStartTag Method

The **RenderStartTag** method is called upon rendering of the start tag in the CSP page. **RenderStartTag** writes code into the routine builder object that renders this element. Text that occurs before the <csr:children> tag in the body of a <csr:action> are written to the routine builder object using a series of different write methods, depending on the type of text to be written.

For example, the following code:

```
<csr:action>
<script language="Cache" runat=server>
   Set myfile="c:\temp.txt"
   Open myfile:("FR":100)
   Use myfile:()
   Read var1
   Close myfile
</script>
</csr:action>
```

results in the following **RenderStartTag** method being generated in the created rule class upon compilation:

```
Method RenderStartTag() As %Status
{
 New element Set element=##this
 Do ..WriteCSPServer(" Set myfile=""c:\temp.txt""",0)
 Do ..WriteCSPServer(" Open myfile:(""FR"":100)",1)
 Do ..WriteCSPServer(" Use myfile:() ",1)
 Do ..WriteCSPServer(" Read var1",1)
 Do ..WriteCSPServer(" Close myfile",1)
 Quit $$$SKIPCHILDREN
}
```

The **RenderStartTag** method can contain other statements, depending on the structure of the rule definition. If the <script runat=compiler> tag was specified in the action, resulting in the creation of **CompilerMethod** methods, the **CompilerMethod** methods are called at the beginning of the **RenderStartTag** method using the following commands (shown for the instance of **CompilerMethod1**):

```
Set %statuscode=$$$OK Do ..CompilerMethod1()
Quit:$$$ISERR(%statuscode) %statuscode
```

In addition to different **Write** methods and calls to **CompilerMethod** methods, the **RenderStartTag** method can also contain other commands depending on whether one or more csr tags were used in the <csr:action> definition.

8.7.2.1 Quit Statement for <csr:children>

If the <csr:children> tag is in the .csr file, then the last line of the generated **RenderStartTag** method is:

```
Quit $$$PROCESSCHILDREN
```

This indicates that the children should be processed upon completion of the **RenderStartTag** method. The **RenderEndTag** method is also written explicitly to the class and writes statements that occur after the <csr:children> tag is called (by default, the **RenderEndTag** method does nothing).

8.7.2.2 Quit Statement for <csr:default>

If the <csr:default> tag is used in the action definition, the generated **RenderStartTag** method contains the following commands:

```
Do ..RenderDefaultStartTag()
Quit $$$PROCESSCHILDREN
```

8.7.3 CompilerMethod[n]() Method

The **CompilerMethod** method is generated from the .csr file if `runat=compiler` was specified for one or more `<script>` tags. It can be called anywhere in the **RenderStartTag** method, depending on the position of the `<script runat=compiler>` statement. Multiple `<script>` tags with compile-time code generate multiple **CompilerMethod** methods (**CompilerMethod1()**, **CompilerMethod2()**, and so on). Unlike the other two methods, compile-time ObjectScript statements that are in the .csr file are copied line for line into the body of this method.

For example, consider the following compile-time code in the .csr rule file:

```
<script language="Cache" runat=compiler>
   SET ^client(2,1,1)=..InnerText()
   </script>
```

When the .csr file is compiled, the following method is generated in the associated rule class:

```
Method CompilerMethod1() [ Language = cache ]
{
    SET ^client(2,1,1)=..InnerText()
}
```

8.7.4 RenderEndTag Method

The **RenderEndTag** method is generated in the rule class if the `<csr:children>` or `<csr:default>` tag is in the .csr file rule definition. It is called upon rendering of the end tag. Any statements that occur after the `<csr:children>` tag are written to the routine builder in this method, similar to the **RenderStartTag** method.

Below is a sample rule definition taken from the `barchart.csr` example on the CSP Samples page. Notice the placement of the `<csr:children>` tag in the table declaration.

```
<csr:rule name="iscbarchart" match="isc:barchart" language="any">
<csr:action>
<table bgcolor='##(..GetAttribute("BGCOLOR"))##' border=0 cellspacing=0
 style='border: ##(..GetAttribute("BORDER","solid blue"))##;'><tr>
<csr:children>
</tr></table>
</csr:action>
</csr:rule>
```

Upon compilation, an iscbarchart rule class is generated, with a call to process the children in the `Quit` statement of the **RenderStartTag** method. The HTML that was present after the `<csr:children>` tag in the rule file is written in the **RenderEndTag** method:

```
Import User

Class csr.csp.iscbarchart Extends %CSP.Rule
{

Parameter CSRURL = "w:/csp/samples/barchart.csr";

Method RenderEndTag() As %Status
{
 New element Set element=##this
 Do ..WriteText("",1)
 Do ..WriteCSPText("</tr></table>",0)
 Quit $$$OK
}

Method RenderStartTag() As %Status
{
 New element Set element=##this
 Do ..WriteText("",1)
 Do ..WriteCSPText(
  "<table bgcolor='##(..GetAttribute(""BGCOLOR""))##' border=0 cellspacing=0"
  ,1)
 Do ..WriteCSPText(
  " style='border: ##(..GetAttribute(""BORDER"",""solid blue""))##;'><tr>"
  ,0)
 Quit $$$PROCESSCHILDREN
```

```
}
}
```

8.8 Using %CSP.Rule Methods

The %CSP.Rule class contains several instance methods available for use in a <csr> rule definition. These methods can be one of two types:

- Methods that are read-only and return the value of an element

  GetAttribute

  QuoteAttribute

  GetAttributesOrdered

  IsDefined

- Methods that modify elements in the Document Object Model

  InnerText

  AddChildElement

  SetAttribute

  OnMatch

8.8.1 GetAttribute Method

8.8.1.1 Syntax

```
GetAttribute(name As %String, default As %String = "")
```

The method **GetAttribute** gets the value of the HTML attribute *name* for this element. The value already has any **##( )##** and **##''##** expressions resolved.

The following example sets the name and size of an HTML grid:

```
    Set tablename=##this.GetAttribute("NAME")
    Set maxcols=##this.GetAttribute("COLS")
    Set maxrows=##this.GetAttribute("ROWS")
```

8.8.2 QuoteAttribute Method

8.8.2.1 Syntax

```
QuoteAttribute(name As %String, default As %String = "")
```

The method **QuoteAttribute** gets the value of the HTML attribute *name* for this element. The value is quoted for substitution with **#()#**, **##( )##** and **##''##** expressions resolved.

The following example is taken from the <csp:loop> tag, which contains four attributes: one of type string (*"counter"*), and three of type integer (*"FROM"*, *"STEP"*, and *"TO"*). It retrieves their values and prints them as strings on the CSP page:

```
<SCRIPT LANGUAGE=Cache RUNAT=COMPILER>
    New counter,from,step,to
    Set counter=..GetAttribute("counter","counter")
    Set from=..QuoteAttribute("FROM",1)
    Set step=..QuoteAttribute("STEP",1)
    Set to=..QuoteAttribute("TO",1)
    Do ..NewBlock()
    Do ..WriteServer(" For "_counter_"="_from_":"_step_":"_to_" {")
</SCRIPT>
```

8.8.3 GetAttributesOrdered Method

8.8.3.1 Syntax

```
GetAttributesOrdered(ByRef paramsordered)
```

The method **GetAttributesOrdered** returns all the parameters in a ordered array.

8.8.4 IsDefined Method

8.8.4.1 Syntax

```
IsDefined(name As %String)
```

The method **IsDefined** indicates whether the HTML attribute *name* is defined.

```
 If ..IsDefined("CAPTION") {
     Do ..WriteServer(setCmd
                      _ " nvp(""CAPTION"") = "
                      _ ..QuoteAttribute("CAPTION"))
 }
```

8.8.5 InnerText Method

8.8.5.1 Syntax

```
InnerText()
```

The method **InnerText** gets the text contained in the start and end tag.

This method can be used to collect and replace the contents of the tag with text specified by the language and domain attributes.

```
 Set %text=##class(%CSP.TagLanguage).GetText(##this,..InnerText())
```

8.8.6 AddChildElement Method

8.8.6.1 Syntax

```
AddChildElement(atom As %CSP.AbstractAtom)
```

The method **AddChildElement** adds a child atom to this element.

8.8.7 SetAttribute Method

8.8.7.1 Syntax

```
SetAttribute(name As %String, value As %String)
```

The method **SetAttribute** sets the HTML attribute *name* for this element to *value*.

The following example sets the *"NAME"* attribute to the default (`"FORM"`) if no `"NAME"` attribute is specified:

```
If ('..IsDefined("NAME")) {
    Do ..SetAttribute("NAME","FORM")
}
```

8.8.8 OnMatch Method

8.8.8.1 Syntax

```
OnMatch(rule As %CSP.Rule) returns %Status
```

The default **OnMatch** method for rules is to do nothing. **OnMatch** may be overridden by user rules.

I need to explain what the method does — in general; how does it check rules to see if they match.

One way to create common code to be inherited by all CSP pages in an application is to override the **OnMatch** method.

The **OnMatch** method for each matching rule is called after the selected rule is added to the rule DOM. When the rule is matched -- if more than one rule matches with the same selectivity then each **OnMatch** will be called. The order should not be depended on as it depends upon compile order and other factors.

The rule class instance passed to the **OnMatch** method is the selected rule that has been put into the DOM; this rule is the rule that had the most specific match (according to rules that are very similar to the XSLT Selectivity rules). The **OnMatch** from all matching rule classes is called using this most selective rule.

The **OnMatch** method can modify the DOM via the rule that is passed in. See the %URL.FORM rule in url.csr that we ship and the attached sample. The sample rule is less selective than the built-in system rule; this allows default behavior to continue. The **OnMatch** callback is designed for this purpose, since it is called when a rule matches, even if it is not the most selective rule. As an alternative to the sample, one could create a custom rule that is added to the DOM and then fixes things up further when the DOM is being transversed during page class generation. This option is possible, but is more complex.

Instead of overriding the **OnMatch** method, you could put the code into your own rule. We do not recommend that you overrivde system rules.

Overriding the system-supplied rules (especially `html`, `head` and `body`) requires great care and dependency on the internals of the rule compiler. InterSystems recommends taking the approach that we take for the /csp/samples CSP pages where we created the `isc:SAMPLE` rule and included it in every page. It is simple to write a routine that loops over existing pages and adds the new custom tag.

8.9 Using <csr> %CSP.AbstractAtom Write Methods

The write methods that are used in the **RenderStartTag** and **RenderEndTag** methods of a rule class are responsible for writing the code created by the rule definition to the CSP page class. This permits the CSP page class to contain the

appropriate commands necessary to exhibit the intended behavior when the page is requested. The %CSP.AbstractAtom class contains the definition for these write methods:

- **WriteText**
- **WriteCSPText**
- **WriteExpression**
- **WriteServer**
- **WriteCSPServer**

8.9.1 WriteText Method

8.9.1.1 Syntax

```
WriteText(line As %String, crlf As %Boolean = 0)
```

The **WriteText** command generates a **Write** command in the CSP page class to write the contents of a line. It takes in two arguments: the string to be written, and a carriage return line feed Boolean indicating whether a newline should be written.

8.9.2 WriteCSPText Method

8.9.2.1 Syntax

```
WriteCSPText(line As %String, crlf As %Boolean = 0)
```

The **WriteCSPText** command generates a **Write** command in the CSP page class to write the contents of a line with the processing of `##()##`, `##''##`, `#server`, `#url`, and `#()#` expressions. It takes in two arguments: the string to be written, and a carriage return line feed Boolean indicating whether a newline should be written. For example, the following line in the body of a `<csr:action>` tag in a .csr rule file:

```
<B><I>##(##this.GetAttribute("VALUE"))##</I></B>
```

generates the following statement in the rule class upon compilation:

```
 Do ..WriteCSPText("<B><I>##(##this.GetAttribute(""VALUE""))##</I></B>",0)
```

8.9.3 WriteExpressionText Method

8.9.3.1 Syntax

```
WriteExpressionText(expr As %String, crlf As %Boolean = 0)
```

The **WriteExpressionText** command generates a write command in the CSP page class to write the text returned by an ObjectScript expression. The text returned should already be properly quoted. It takes in two arguments: the string to be written, and a carriage return line feed Boolean indicating whether a newline should be written.

8.9.4 WriteServer Method

8.9.4.1 Syntax

```
WriteServer(line As %String, keepTogether As %Boolean = 0)
```

The **WriteServer** command generates an ObjectScript command in the CSP page class that is in line. It takes in two arguments: the string to be written, and a Boolean indicating whether to append the string to the previous statement.

8.9.5 WriteCSPServer Method

8.9.5.1 Syntax

```
WriteCSPServer(line As %String, keepTogether As %Boolean = 0)
```

The **WriteCSPServer** command generates an ObjectScript command in CSP page class that is in line with **##()##**, **#()#**, and ##''## resolved. It takes in two arguments: the string to be written, and a Boolean indicating whether to append the string to the previous statement. For example, the following ObjectScript code in a .csr rule file:

```
<script language="Cache" runat=server>
   Set myfile="c:\temp.txt"
   Open myfile:(80:"C":"|")
   Use myfile:()
   Read ^client(3,1,1)
   Close myfile
</script>
```

generates the following statement in the rule class upon compilation:

```
 Do ..WriteCSPServer(" Set myfile=""c:\temp.txt""",0)
 Do ..WriteCSPServer(" Open myfile:(80:""C"":""|"") ",1)
 Do ..WriteCSPServer(" Use myfile:()",1)
 Do ..WriteCSPServer(" Read ^client(3,1,1)",1)
 Do ..WriteCSPServer(" Close myfile",1)
```

8.10 Using <csr> %cspQuote Methods

The %cspQuote routine definition contains definitions for two different quoting methods.

- **Quote**
- **QuoteCSP**

8.10.1 Quote Method

8.10.1.1 Syntax

```
$$Quote^%cspQuote(line As %String)
```

Surrounds the input string with quotes.

8.10.2 QuoteCSP Method

8.10.2.1 Syntax

```
$$QuoteCSP^%cspQuote(line As %String)
```

Surrounds the input string with quotes and resolves **#()#**, **##()##**, ##''##, **#server**, and **#url** calls.

8.11 Creating a <grid> Tag to Display a Table

This section contains an example of a rule, called `GridExample`, that creates two tags that create a table of information on a CSP page.

- `<GRID>` tag
- `<GRIDDATA>`tag

8.11.1 Grid Rule Definition

```
<csr:rule name="GridExample" match="GRID">
<csr:action>
<script language =Cache runat=compiler>
    Set maxrows=##this.GetAttribute("COLS")
    Set maxcols=##this.GetAttribute("ROWS")
    Do ..WriteText("<TABLE>",1)
    Set GRIDDATA=""
    ;Get Grid Data
    Set count=##this.Children.Count()
    For i=1:1:count {
         Set el=##this.Children.GetAt(i)
         Set tagname=el.TagName
         If tagname="GRIDDATA" {
             Set value=el.GetAttribute("VALUE")
             Set col=el.GetAttribute("COL")
             Set row=el.GetAttribute("ROW")
             Set GRIDDATA(row,col)=value
         }
    }
    ; Write Grid Elements
    For row=1:1:maxrows {
        Do ..WriteText("<TR>",1)
        For col=1:1:maxcols {
            Set d=$G(GRIDDATA(row,col))
            Do ..WriteCSPText("<TD>"_d_"</TD>",1)
        }
    }
    Do ..WriteText("</TR>",1)
    Do ..WriteText("</TABLE>",1)
</SCRIPT>
</csr:action>
</csr:rule>
```

Although the *<GRIDDATA>* attributes are handled in the rule definition for `<GRID>`, an empty rule is still necessary to instantiate the `<GRIDDATA>` tag:

```
<csr:rule name="GridDataExample" match="/GRID/GRIDDATA">
<csr:description>
This purpose of this empty rule is to instantiate the GRIDDATA tag
into the Document Object Model.
</csr:description>
<csr:action>
<csr:default>
</csr:action>
</csr:rule>
```

8.11.2 Generated Grid Class

The above rule definitions compile into the following two classes:

- Grid
- GridData

8.11.2.1 Grid Class

```
Import User

Class csr.csp.GridExample Extends %CSP.Rule
{

Parameter CSRURL = "/csp/user/GRIDEXAMPLE.CSR";

Method CompilerMethod1() [ Language = cache ]
{
     Set maxrows=##this.GetAttribute("COLS")
     Set maxcols=##this.GetAttribute("ROWS")
     Do ..WriteText("<TABLE>",1)
     Set GRIDDATA=""
     ;Get Grid Data
     Set count=##this.Children.Count()
     For i=1:1:count {
         Set el=##this.Children.GetAt(i)
         Set tagname=el.TagName
         If tagname="GRIDDATA" {
             Set value=el.GetAttribute("VALUE")
             Set col=el.GetAttribute("COL")
             Set row=el.GetAttribute("ROW")
             Set GRIDDATA(row,col)=value
         }
     }
     ; Write Grid Elements
     For row=1:1:maxrows {
         Do ..WriteText("<TR>",1)
         For col=1:1:maxcols {
             Set d=$G(GRIDDATA(row,col))
             Do ..WriteCSPText("<TD>"_d_"</TD>",1)
         }
     }
     Do ..WriteText("</TR>",1)
     Do ..WriteText("</TABLE>",1)
}

Method RenderStartTag() As %Status
{
 New element Set element=##this
 Set %statuscode=$$$OK Do ..CompilerMethod1()
 Quit:$$$ISERR(%statuscode) %statuscode
 Quit $$$SKIPCHILDREN
}

}
```

8.11.2.2 GridData Class

```
Import User

Class csr.csp.GridDataExample Extends %CSP.Rule
{

Parameter CSRURL = "/csp/user/GRIDEXAMPLE.CSR";

Method RenderEndTag() As %Status
{
 New element Set element=##this
 Do ..RenderDefaultEndTag()
 Quit $$$OK
}

Method RenderStartTag() As %Status
{
 New element Set element=##this
 Do ..RenderDefaultStartTag()
 Quit $$$PROCESSCHILDREN
}

}
```

8.11.3 Using the Grid Rule

The grid rule can now be used in the body of a CSP page:

```
<html>
<head>
<title>Grid Example</title>
</head>
<body>
<grid cols="5" rows="5">
<griddata value="Cell-1-1" col="1" row="1">
</griddata>
<griddata value="Cell-2-1" col="2" row="1">
</griddata>
<griddata value="Cell-2-2" col="2" row="2">
</griddata>
<griddata value="Cell-2-3" col="2" row="3">
</griddata>
<griddata value="Cell-2-4" col="2" row="4">
</griddata>
<griddata value="Cell-2-5" col="2" row="5">
</griddata>
<griddata value="Cell-3-1" col="3" row="1">
</griddata>
<griddata value="Cell-4-1" col="4" row="1">
</griddata>
<griddata value="Cell-4-3" col="4" row="3">
</griddata>
<griddata value="Cell-5-1" col="5" row="1">
</griddata>
<griddata value="Cell-5-5" col="5" row="5">
</griddata>
</grid>
</body>
</html>
```

8.11.4 Grid Rule Displayed Page

The CSP page now displays the following:

```
  Cell-1-1 Cell-2-1 Cell-3-1 Cell-4-1 Cell-5-1
           Cell-2-2
           Cell-2-3          Cell-4-3
           Cell-2-4
           Cell-2-5                    Cell-5-5
```

A

CSP Error Notes

This chapter describes causes and approaches to solving selected CSP errors.

Table I–1: CSP Error Codes, Error Messages, and When Reported

Error Code	Error Message	When Reported
5902	Rule '%1' does not exist	Reported when calling `%apiCSP` to add attributes to a rule if you specify a rule name that does not exist.
5903	Rule name is required	Reported if you attempt to add or delete a rule but do not provide a name for the rule.
5904	Attribute '%2' is required for tag '<%1>' on line number %3	Reported if you did not supply a required attribute for a tag in the CSP page. The page cannot compile without this required attribute.
5905	The value of attribute %1, '%2', is invalid, on line number %3	Reported if the value of an attribute in a CSP page is not a valid choice. For example if you define `<script language="Cache" runat="XXXXX">`, the `runat` value is not a valid choice. The CSP compiler cannot compile this page and reports this error.
5906	Session ID is missing	Reported if you attempt to create an instance of %CSP.Session without supplying a session ID in the **%New** method. For example, `Set session=##class(%CSP.Session).%New()` reports this error but `Set session=##class(%CSP.Session).%New(1234)` does not as it passes the session ID `1234`.
5907	Session ID '%1' does not exist	Reported if you attempt to load an existing **%CSP.Session** but pass the %OpenId a session ID that is not stored in Caché.
5908	Failed to create class '%1': %2	Reported by the CSP compiler if it cannot create the class corresponding to the CSP page.
5909	There is no closing tag for the tag <%1> on line number %2	Reported if the CSP compiler detects that you opened a tag but never closed it (if the tag specifies that it needs a closing tag in the rule definition).

Error Code	Error Message	When Reported
5911	Character Set '%1' not installed, unable to perform character set translation	Reported if the character set specified in the CSP page to output this page is not installed in Caché. This could be the character set specified in the %response.CharSet property in the **OnPreHTTP** method or specified in the page using the `<csp:content charset="xxx">` tag or the `<meta http-equiv="Content-Type" content="text/html; charset=xxx">`. See the charset property of the class %CSP.Page. Check that you intend to use the character set reported in the error and if so, check that this is installed in Caché. or by setting the %response.CharSet property in the **OnPreHTTP** method.
5912	CSP Page '%1' does not exist	Reported if you request a CSP page that does not exist. You may have mistyped a URL or a link on another CSP page may be incorrect. Check if the page exists on the server and, if not, then look for where the link came from. If the page should exist, make sure the CSP application settings are correctly set to point to the right directory and check that the CSP file exists on the disk. This error only occurs if the `autocompile` option is on and the CSP engine tries to compile this page and cannot find the file.
5914	CSP Application '%1' does not exist	Reported when the application part of the URL cannot be found in the CSP application list. For example, you try to load the page /cspx/samples/menu.csp with a type of `cspx` rather than `csp`, then Caché cannot find the CSP application. Check the list of applications by navigating to **System Administration** > **Security** > **Applications** > **Web Applications** in the Management Portal and check the command for mistakes.
5915	Cannot allocate a license	Reported if the license limit has been reached so this new request for a CSP session cannot be granted. You may be able to reduce the default timeout on CSP sessions specified in the CSP application configuration or you need to look at buying more licenses.
5916	Illegal CSP Request	Reported when you try to reach a private page by entering the URL instead of being redirected from another CSP page which includes the encrypted token to allow access to this page, or by using an invalid encrypted token to allow access to this private page.
5917	HTTP method '%1' not supported by CSP	Reported when you attempt to use an unsupported HTTP method. HTTP methods supported are GET, POST, HEAD. We do not support other HTTP methods in the CSP server at present. It can also be caused by an incompatible version of the CSP gateway talking to the CSP server.

Error Code	Error Message	When Reported
5918	You are logged out, and can no longer perform that action	Reported if the CSP request contains encrypted data, but the session is a brand new session, so there is no way that the decryption key can match the encrypted data. Typically this is because the session has timed out. Then the user subsequently does something in the browser to cause another request. You can increase the session timeout value or use the error mechanism to redirect the user to an initial page so they can start their action again.
5919	The action you are requesting is not valid	Reported typically when passing an encrypted string to Caché from the CSP page where the decryption key does not match the key used to encrypt this data. This can be caused by the user tampering with the URL manually or by anything that could change the value of the encypted string between it being generated in Caché and returned back to Caché in the next HTTP message.
5920	Must run this CSP page from namespace '%1'	Each CSP application is tied to a specific namespace in Caché. This error is reported if you attempt to do something such as compiling a page from /csp/samples/loop.csp in the USER namespace when the /csp/samples application is tied to the SAMPLES namespace.
5921	The CSP application '%1' must specify a namespace to run in	Reported if the configuration of the CSP application is missing the namespace. This generally indicates that the CPF file has been badly edited by hand as the Management Portal does not allow a CSP application to be created without a namespace.
5922	Timed out waiting for response	Reported by the %Net.HttpRequest object when it times out waiting for a response from the HTTP server it is talking to.
5923	Redirected %1 times, appears to be a redirection loop	Reported If more than 4 redirects are detected in one page. The compiler assumes that there is a loop. If a CSP page uses the **ServerSideRedirect** to jump to another page there is a possibility that page A.csp could redirect to B.csp which redirects to A.csp creating a loop.

Error Code	Error Message	When Reported
5924	An error occurred and the specified error page could not be displayed - please inform the web master	When an error in a CSP page occurs at runtime, the CSP engine redirects to a user-specified error page that can handle the error in any manner it wishes. If, however, this user-specified error page does not exist or there is an error in generating this error page, then the CSP engine logs the fact that something has gone wrong using `BACK^%ETN` and reports this error message. As this error may appear on a production system if there is a bug in the user— written error page, the message is deliberately vague. To resolve this error, first check that the error page specified in the CSP application exists and then look at possible bugs in this error page. See also a question on this in the FAQ appendix.
5925	<SCRIPT LANGUAGE=Cache> tag is missing either RUNAT or METHOD attribute, on line number %1	Reported if the `<script language="Cache">` tag is missing the required attribute *runat* (to tell the CSP compiler when this code should run), or the*method* attribute to create a new method.
5926	Unable to redirect as HTTP headers have already been written and flushed	Reported if you try to use a server side redirect after data has been written to the browser .If you attempt to use the *%response.ServerSideRedirect* feature to redirect to another page, this must be done before any data has been written back to the browser. Typically this means you must do this in the **OnPreHTTP()** method of the page.
5927	Unable to load page '%1' because its class name conflicts with the class '%2' that is already loaded	Reported if you have two CSP files with identical names in different applications in the same namespace: For example, if you have two CSP applications, `/test` and `/anothertest`, both in the USER namespace. which are in different directories on the Caché server, each of which has a file test.csp. If you have `autocompile` turned on and you enter the URL /test/test.csp the CSP compiler compiles this page into the class csp.test. If you enter the URL /anothertest/test.csp, it tries to load this page to create the class csp.test, finds it already exists for a different application and reports this error. If it did not do this, you would see very poor performance as each request would recompile the entire page. Either avoid using identical file names in the same namespace or change the package defined in the CSP application, which defaults to `csp`. For example, change`/anothertest` to use package name package. Then when it compiles test.csp, it creates the class name package.test which does not conflict with the other application that uses csp.test.

Error Code	Error Message	When Reported
5931	Can only call this method/set this value in OnPreHTTP() before page has started to be displayed	Reported if you call a function that needs to be called in the **OnPreHTTP()** method of the page so that it can modify some parameters before any data is output to the browser. Move this call to the **OnPreHTTP** method to resolve this.
5932	Action not valid with this version of the CSP Gateway on the web server	Reported if the version of the CSP gateway you are using does not support this action. Either do not use this feature or upgrade the version of the CSP gateway to a later version.
5933	The CSP server had an internal error: %1	Reported if an unexpected error condition has occurred inside the CSP engine. Please report this to InterSystems support.
5954	Failed to lock CSP page.	When a CSP page is autocompiled it is first locked to make sure that two jobs do not both attempt to compile the same page at the same time. If the lock is not released by the other job in 60 seconds, it assumes the compile failed for some reason and reports this error message. Try recompiling this page from Studio to see if any errors are reported.
5955	CSPAppList query: invalid data in Fetch().	Reported if the query to determine the list of CSP applications is invalid. This error should never be seen on a working system.
5956	Directory '%1' for CSP Application '%2' does not exist	Reported if the directory pointed to by the CSP application does not exist in the file system.
5961	Unable to convert character set '%1'.	Reported when a request from a browser comes in. The information sent by the browser is converted into the current Caché default locale and there is an error. To debug the conversion, isolate the information being sent by the browser and convert it from that character set manually in a test program.
5962	Unable to allocate new session.	Reported when calling **%session.ForceNewSession()** if there are no new slots in this session Id.
5963	Invalid SysLog level: %1.	Reported when setting the internal log level if the level is outside the allowed range.

B

Localization and Tag-Based Development

This appendix discusses how to localize CSP pages when you are doing tag-based development. Also read the chapter "Localizing Text in a CSP Application."

B.1 Introduction

During tag-based development of CSP pages, you can configure certain tags so that they substitute a message dictionary entry for what would otherwise be literal text. Do this by providing the localization attributes *language*, *domain*, or *textid* inside the tag. The following tags support these attributes:

- <csp:text>
-
-
- <input> (when the *type* is "SUBMIT", "BUTTON", or "RESET")

For the most part, these tags work only at runtime, when the values provided for *language*, *domain*, and *textid* indicate which message to retrieve from the message dictionary.

However, in limited cases these tags serve different purposes at compile time and runtime. They can automatically generate entries for the message dictionary at compile time, then retrieve and display these entries at runtime. The following sections explain how this works:

- Localization Tags at Runtime
- Localization Tags at Compile Time

For a simple demonstration of a localized CSP application, enter the following URL while Caché is running: http://localhost:57772/csp/samples/language.csp.

B.2 Localization Tags at Runtime

Important: This topic applies to the CSP tags <csp:text>, , , <input>, and `<button>` only.

At runtime, when the CSP page displays, the tag replaces itself and its contents with text from the message dictionary. The choice of text is specified by the *language*, *domain*, and *textid* attribute values supplied by the tag.

For example, the following syntax is replaced by the message specified by the `fr` (French) language, `sample` domain, and `menu` message ID. The text provided inside the `<csp:text>` tag (`Menu` in this case) is ignored:

```
<csp:text textid="menu" language="fr" domain="sample">Menu</csp:text>
```

Defaults for *language*, *domain*, and *textid* are available if any of the attributes are omitted or empty (value `""`):

- If *language* is not specified, the value of %response.Language is used.
- If *domain* is not specified, the value of %response.Domain is used.
- *textid* is required (with exceptions; see the section "Localization Tags at Compile Time")

B.2.1 Default Language

Most developers intentionally provide no *language* attribute so that, at runtime, the language defaults appropriately for the locale. If not supplied, *language* defaults to the value of %response.Language, which automatically takes its runtime value from the browser settings.

You can see examples of this convention in the source code for the localization example in the SAMPLES namespace. View it as follows:

1. Start Studio.
2. Change to the SAMPLES namespace.
3. In the Workspace window, click the Namespace tab.
4. Choose **CSP Files**.
5. Open the file /csp/samples/language.csp.

B.2.2 Default Domain

The %response.Domain property is initialized with the value of the DOMAIN parameter of the CSP page class. You can also set a value for %response.Domain using the `domain` attribute of the `<csp:class>` tag, as in:

```
<csp:class domain="myDomainName">
```

B.2.3 Message Arguments

If the message text contains arguments (*%1*, *%2*, *%3*, *%4*) the following tag attributes let you specify the corresponding substitution text: *arg1*, *arg2*, *arg3*, *arg4*. You may provide literal values or use variables. For example:

```
<csp:text textid="sessionInfo" arg1="#(userName)#" arg2="#(roleID)#" />
```

B.2.4 Button Text

In tag-based CSP files, the text displayed on a button is normally specified using the *value* attribute of the `<input type="button">` or `<button>` tag.

When an `<input>` tag uses localization attributes (*language*, *domain*, or *textid*) the *value* attribute is ignored. The text displayed on the button is the text from the message dictionary. If you want to localize the text on a `<button>` tag, use the *language*, *domain*, or *textid* attributes of the `<csp:text>` tag.

B.3 Localization Tags at Compile Time

Important: This topic applies to the CSP tags <csp:text>, , , and <input> tags only.

The *textid* attribute may have the empty value "". If so, when you compile the tag-based CSP file a new message is automatically generated in the message dictionary. The text for the generated message consists of the contents of the CSP tag. Caché generates a message ID by calculating the 32–bit CRC (Cyclic Redundancy Check) of this text.

Only the <csp:text> tag permits you to actually omit the required *textid* attribute. The other localization tags require you to at least provide an empty value "".

If a <csp:text> tag lacks a *textid* attribute, the system automatically generates a new message and message ID value. In cases where you omit *textid* from <csp:text>, the text inside the tag may include an optional @*textID*@ string at the beginning. *textID* is the message ID that you wish to assign to the message. For example:

```
<csp:text>@simpleMenu@Menu</csp:text>
```

In the above example, Caché does not generate the message ID. It generates a message whose text is Menu and gives it the message ID simpleMenu.

When a CSP tag has been used to generate a message dictionary entry at compile time, it still works as a reference to retrieve that entry at runtime. See the section "Localization Tags at Runtime."

C

Frequently Asked Questions About CSP

Troubleshooting

How do I fix a Zen error about zenutils.js or other js file?

The web server must be configured to serve .js files through the CSP Gateway. This is not done by the Caché installer, even when the option to configure the external webserver is selected, due to security concerns. For details, see CSP Gateway Configuration Guide. Find the section for your operating system and your configuration option. Then within your option, find one or both of the sections (depending on the option) called "Mapping the CSP File Extensions" and "Registering Additional File Types with CSP."

On an Apache web server, add the following to the httpd.conf file (there are a few other ways, but this is easiest):

```
<Location /csp>
CSP On
SetHandler csp-handler-sa
</Location>
```

On IIS7, do the following:

1. Open the Control Panel. Select **Admin Tools** > **IIS Manager**.
2. On menu at left, expand *yourcomputername* > **Sites** > **default website** and click **csp**
3. On the right side of the screen, double-click **handler mappings**.
4. Scroll through the existing mappings, and make sure there is one with *.js in the path column.

If *.js is not there, you need to add it, as follows:

1. Click **Add Module Mapping** .

   Request Path: *.js
   Module: CSPms
   Name: CSP_js (or whatever else you want – any unused name works)

2. Click **Request Restrictions.**

   Mapping: Uncheck Invoke handler only if request is mapped to
   Verbs: Select All verbs
   Access: Select Script

3. Repeat the above steps for *.gif, *.jpg, and *.svg.

On IIS6, do the following:

1. Open the Control Panel. Select **Admin Tools** > **IIS Manager**.
2. On menu at left, expand *yourcomputername* > **WebSites** > **default website** and click **csp**
3. Right-click **CSP** > **properties**.
4. On the **Virtual** tab, click **Configuration**.
5. Scroll through the list, and make sure that *.js is there.

If *.js is not there, you need to add it, as follows:

1. Click **Add Module Mapping** .

   Request Path: *.js
   Module: CSPms
   Name: CSP_js (or whatever else you want – any unused name works)

2. Click **Request Restrictions**.

   Mapping: Uncheck Invoke handler only if request is mapped to
   Verbs: Select All verbs
   Access: Select Script

3. Repeat the above steps for *.gif, *.jpg, and *.svg.

How do I debug a CSP page?

Use the built-in Studio Debugger described in the chapter Using the Studio Debugger in the book *Using Studio*.

Don't set breakpoints with **Debug > View Breakpoints**, as this seems error prone.

1. To debug CSP pages, you must check the option **Tools > Options > Compiler > Keep Generated Source Code**.
2. Open your Workspace window and add your CSP pages to the CSP folder, if they are not already there.
3. Compile your CSP and click the **View Other Code** icon in the toolbar (or select **View > View Other Code**). This lets you see the .cls and .int files. For example, the file A.CSP generates CSP.A.CLS and CSP.1.INT.
4. In the .cls or .int file, right click the line of code where you want a breakpoint and select **Debug > Breakpoints > Toggle Breakpoint** (or select the line and press F9).
5. Select **Debug > Debug Target > ZEN or CSP page**. From the dropdown, select the target CSP page on which the debugger will run and click **OK**. (If you dragged your CSPs to the Workspace window, they appear in the dropdown list.)
6. Click **Go** on the Debug toolbar.

For example, if you have a flow: A -> B; that is, display page A and then follow a link to page B. And there is a bug in page B, you would do the following:

1. Check that A.CSP and B.CSP are in the Workspace window.
2. Compile both.
3. Select **View > Toolbars > Debug** to open the Debug toolbar.
4. Select **Debug > Debug Target > ZEN or CSP page.** From the dropdown, select A.CSP and click **OK**.
5. Open B.CSP and select **View > View Other Code** to open csp.B.CLS.
6. In csp.B.CLS, right click the first line in the OnPageBODY() method and select **Debug > Breakpoints > Toggle Breakpoint**.
7. Click **Go** on the Debug toolbar.
8. Page A is displayed.

9. Click the link that brings up page B.
10. The debugger stops on the breakpoint in B

Why does the following code not compile?

```
<script language="Cache" runat="server">
    write "<script language=javascript>", !
    write "int x = 10; alert(x);", !
    write "</script>"
</script>
```

When compiling a CSP page with `runat="server"` in a script tag, the compiler runs the ObjectScript and converts it into HTML to display on the page. However, after encountering a `<script language="cache" runat="server">` tag, it looks for the `</script>` end tag to signify the end of the ObjectScript code, which, in this case, it finds in the `write` statement. To get this to compile, break the `</script>` tag into two `write` statements:

```
<script language="Cache" runat="server">
    write "<script language=javascript>", !
    write "int x = 10; alert(x);", !
    write "</script",">", !
</script>
```

When I use `&js<alert(“Error!”);>` to display an alert box, the text `alert(“Error!”)` is displayed instead of an alert box. Why?

It is possible that you put this line inside a `runat=“server”` code section or inside a method called from a `runat=“server”` block. To execute JavaScript as the page loads, add the `<script language=“javascript”>` tag as shown in the previous answer.

The code `&js<alert(“Error!”)>` works inside a server side method called via a JavaScript event from the loaded page.

How can I include ObjectScript variables in an alert message?

Use `#()#` syntax. From inside your ObjectScript method, try something like this:

```
 s error = "Bad password"
 &js<alert(#(..QuoteJS(error))#);>
```

The **QuoteJS** method provided by the %CSP.Page class returns a properly quoted JavaScript string. It also correctly escapes any quotes contained within the returned result.

I am getting the following error: *HTTP Request failed. Unable to process hyperevent.* What does this mean?

Hyperevent errors occur when the browser tries to communicate with the CSP broker applet but cannot. This could be related to the code or your configuration. To decide if the problem is your code, load http://localhost:57772/csp/samples/zip-code.csp (where `57772` is the Caché web server port number. Replace 57772 with the actual web server port, if necessary, or leave it out if you have an external web server running (http://localhost/csp/samples/zipcode.csp), or if Caché is installed remotely, replace `localhost` with the IP address or machine name.)

On the zip code page, click **#server**, and enter a zip code, such as `02142`, in the **Zip** box and press **Tab**. If you do not receive a hyperevent error, you are properly configured and your hyperevent error is likely caused by a coding error.

If the problem appears to be coding related, there are two things to look for. Never use **#server** calls as the page is loading. This includes calling them in the **OnLoad** event of the `<body>` tag, either directly or from a JavaScript method called in

OnLoad. It also includes calling them from inside an &js<> line of a runat="server" code block. If you need to call a method as the page loads, use ObjectScript syntax inside a runat="server" block.

```
<script language="cache" runat="server">
    // if method is located in the page or in a class the page inherits from
    d ..mymethod()
    // if class cannot be called using dot syntax
    d ##class(User.MyClass).mymethod()
</script>
```

I received an error message that suggests a problem in a line of my routine, but I can't find the INT routine. Where is it?

Depending on your current settings, you might not keep the source code when you compile a CSP page. To force Caché to maintain this source code, you can do one of two things:

- Compile your CSP pages from the Studio with Keep Generated source code selected. In Studio, click **Tools > Options > Compiler > General Flags**. Select the **Keep generated source code** check box. Then, to compile your CSP page, click **Build > Compile**.
- Compile your CSP pages from the Terminal using the k flag to Keep generated source code. From the Terminal, verify you are in the correct namepace. (If not, change namespaces by entering: **Zn** "*<namespace>*".) To compile your CSP page, enter: **Do** $System.CSP.LoadPage("/csp/*<namespace>*/*<pagename>*.csp", "ck"). For example:

  ```
  Do $System.CSP.LoadPage("/csp/samples/james.csp", "ck")
  ```

To find the generated source code:

Open Studio and verify that you are in the correct namespace. Click **File > Open**. In the **Files of type** list, click **Intermediate Routine (*.int)**. Find csp.<csp_page_name>.x.INT, with *x* being the number of this routine in the series. Large CSP pages are broken into multiple INT routines. The number of the file containing the error is shown in the error message you originally received.

When I run my CSP page, #(variable)# shows up in the browser. Why isn't this being replaced with the data in the variable?

This indicates that your CSP page has not been properly parsed by the Caché CSP Compiler. Ensure that you are running your pages through your web server as follows: http://localhost/csp/<namespace>/page.csp.

Why am I getting Invalid Character error messages when I try to load my CSP page?

If you are *not* loading your page through your web server, this error is common because the browser does not know how to represent your CSP syntax. If your URL looks something like C:/install-dir/csp/user/mypage.csp, you are not loading through your web server.

Your URL should be something like http://localhost:57772/csp/user/mypage.csp or http://localhost/csp/user/mypage.csp.

These messages can also result from **#server()#** calls which are not correctly translated to an ObjectScript method call. From your browser, right-click and view the source of your page. If the source still contains #server, your syntax may be incorrect. Ensure that it is properly formed as: #server(..methodname())#.

If you are passing strings to the method, they must be inside double quotes (" "). Once your CSP page is compiled into HTML, all **#server()#** calls are translated into a call to **csprunservermethod()**.

Why aren't my CSP tags being parsed?

Your web server is not properly passing your CSP page to the CSP Gateway for parsing.

Does CSP log errors? How can I increase logging and where does the log exist?

- If there is an internal error, such as an error in your custom error page, it is logged to the BACK^%ETN error log. If you receive internal errors in this log that are not related to your custom error page, it may be a problem with the core CSP engine and you should contact the InterSystems Worldwide Response Center (WRC).
- Other errors are logged by calling the user-defined error page where the user can add their own logging function.

When I try to load a CSP page, the following error appears: *ERROR #5924: An error occurred and the specified error page could not be displayed - please inform the web master.* What does this mean and how can I solve it?

This error can result from a number of different issues. View the error log to get more specific information about the actual error that occurred. In the Terminal, issue the following command:

```
d ^%ER
```

To view the resultant error log, navigate to **System Operation** > **System Logs** > **Application Error Log** in the Management Portal and check the errors for the appropriate namespace. The errors are contained in folders by date.

If you set up a custom error page, this could mean that your custom error page has no mechanism to deal with an error in the page you are calling. It could also mean that your custom error page itself generated an error. One way to trace this error is to temporarily turn off your custom error page and attempt to load the CSP page.

If your CSP pages work locally, but *not* when called from another computer, it may be that you have a Single User version of Caché or do not have a Caché license. Calling CSP pages from a remote machine requires both a full version of Caché and a valid key with licenses available. Adding a Caché key to a version you have downloaded from the Internet does not give it full functionality. You still need to receive a full version. See also Error 5924 in Appendix B Error Notes

I try to display a CSP page and nothing comes up at all, or I get a login screen, enter a valid username/password, and it will not let me in. What is wrong? I am pretty sure that the CSP gateway is configured to talk to the correct Caché instance, or I am using private Apache install that came with Caché so it is preconfigured.

Make sure that auditing of security events is turned on from the %SYS namespace. At the very least, audit Login, Logout and LoginFailure.

1. In the terminal, in the %SYS namespace, enter `Do ^SECURITY`.
2. Select `Auditing setup`, `Configure auditing events`, and `Create audit event`.
3. Enter `%SYSTEM/%Login/Login`, `%SYSTEM/%Login/Logout`, and `%SYSTEM/%Login/LoginFailure`.

Try accessing the CSP page again and check the audit log to see if you can see any failures. Often this will tell you what the problem is - such as a disabled service, an incorrect password, or that you do not have permission to run this CSP application. For more information on auditing, see the chapter "Auditing" in the *Caché System Administration Guide*.

Sessions and Licenses

Why do I have to log in so often?

In previous releases, when applications shared a session, they could share authentication and data only via the session object.

There are two ways to share a session:

- Via the session cookie path.
- Putting CSPSHARE=1 in the link to the application page.

When the session times out, it is destroyed and its authentication is lost. If an existing page is reloaded, the user has to login again. For applications connected via the session cookie path, they are 'logged in' also, because when you go to a page from one of those applications you get the newly logged-in session.

Not so when sessions are shared via CSPSHARE=1. For example, start an application called SMP which is logged into session X. Now click a link to another application called EMP. That link contains CSPSHARE=1. EMP does not have to log in as it is put into authenticated session X. After a while session X times out and is destroyed. SMP and EMP are without sessions.

Now click a tab containing a page from SMP and, as before, we are asked to log in again. SMP is now in authenticated session Y. We then click a tab containing a page from EMP. There is now no connection between SMP and EMP. EMP is asked to log in again and is put in authenticated session Z.

CSPSHARE is a very fragile way to try to share sessions and is easily severed. Once severed, multiple logins can ensue.

In this release: Use session-sharing only if you decide that data must be shared via the session object. If you need only authentication sharing (and not data sharing), use other options.

Session—Sharing : If you require session sharing, it is best to name all applications with the same Session Cookie Path (which now must be an exact match). You may have to rename your applications, such as, /csp/sys/tool/smp and /csp/sys/tool/emp.

If you require session sharing and you can’t name all applications with the same Session Cookie Path, then use the CSP-SHARE element. However, previous idiosyncrasies, such as multiple logins after time out, will contain to manifest themselves.*Use CSPSHARE as a last resort.*

Authentication-Sharing: If the design calls for sharing authentication information, but does not require sharing session data, use one of the new authentication features.

- Login Cookies [Share Authentication when Enter Application]

  Login Cookies hold information about the most-recently logged in user. When they are enabled, a newly— accessed application tries to use that authentication.

  For Login Cookies, each application is in a separate session. These sessions are independent once that session has been authenticated. So logging out or timing out in one session does not affect the other sessions.

  Unauthenticated logins are not saved in the Login Cookie. If application A logs in to user Q, then application B as unauthenticated, then application C uses login cookies, application C will be logged in as Q.

- Group-By-Browser [Share Authentication Continuously]

  If you want a group of applications to act as an authentication cluster, then use group-by-browser.

  All applications stay in authentication sync. If one logs out, they are all logged out. If one logs into user Q, they are all logged into user Q. (The only exception is that if any applications are unauthenticated, they are treated as pariahs and ignored as far as authentication is concerned.)

How do I end a CSP session?

To end a CSP session, set the %session.EndSession property to 1 in an ObjectScript method. If your CSP application times out, the session is ended automatically by your CSP class.

I closed my CSP session, but Caché still reports that I am using a license. Why?

If you have visited only a single page and you logout or the session times out, CSP provides a *grace period* of 5–10 minutes, during which it reserves the license for you, so that you can recapture the same license if you return quickly. The grace period is the longer of:

- 5 minutes from the end of the session (a timeout or a logout) or
- the amount of time that would ensure 10 minutes from when the session started (ensuring a minimum session of 10 minutes)

The following table summarizes how and when licenses are released:

Case	User has visited one page	User has visited multiple pages
The code explicitly sets %session.EndSession to 1 (for example, when the user clicks **Logout**)	The session receives the grace period. The license is released when the grace period expires.	The license is released immediately
The browser is open and the session has not timed out	The license is retained	The license is retained
The user closed the browser but the session has not timed out	The license is retained	The license is retained
The session has timed out	The session receives the grace period. The license is released when the grace period expires.	The license is released immediately

Here are some examples of how the grace period works when you have visited a single page:

- The user logs in at 12:00 and logs out at 12:15. The grace period is 5 minutes, so the license is free at 12:20.
- The user logs in at 12:00 and logs out at 12:03. The minimum license use time is 10 minutes, so the license is free at 12:10.
- The user logs in at 12:00 and closes the browser at 12:10. The timeout is set to 15 minutes, so the session ends at 12:25. The grace period is 5 minutes, so the license is free at 12:30.

How can I change the timeout for my applications?

The default timeout on applications is set in each namespace to 900 seconds (15 minutes).

- To change the Timeout for all CSP pages within a certain namespace:
  1. From the Caché cube, click **Management Portal**. Log in, if necessary.
  2. On the main page of the Management Portal, navigate to **System Administration** > **Security** > **Applications** > **Web Applications**.
  3. On the **Web Applications** page, click **Edit** for the CSP application to configure.
  4. In the **Default Timeout** field, enter a new value (in seconds) and click **Save**.

- To change the Timeout for a specific application, put the following inside your page, where *x* is the timeout value in seconds.

```
s %session.AppTimeout = x
```

I want to perform cleanup or logging when a user CSP session times out. How can I do this?

1. Create an event class with an **OnTimeout** class method.
2. Specify this as the event class for your application in one of the following ways:
   - In the Management Portal, navigate to **System Administration** > **Security** > **Applications** > **Web Applications**, click **Edit** for the CSP application to configure. In the **Event Class** field, enter the class name to use, such as User.MyEventClass.
   - In your CSP page, use the %session.EventClass property:

```
<script language="cache" runat="server">
s %session.EventClass = "User.MyEventClass"
</script>
```

3. In your **OnTimeout** method, log any information you wish to keep.

   Note: At this point you cannot send information back to the browser (alerts or redirects).

How can I pass information between pages?

There are a number of ways to pass information:

- Put the information in the links to the next page as additional parameters. These are accessible from the %request object:

```
http://myserver/csp/user/mypage.csp?id=3&name=bill
```

  To access the information, use **`%request.Get("id")`**.

  To display the information directly on your page, use **`#(%request.Get("name"))#`**.
- Use the %response.Context multidimensional property to define a set of name-value pairs that are automatically added to all links and forms by the CSP engine.
- Put the data in the %session object. This can cause problems if the user opens the application in two browsers at the same time.

I want to forward the user to another web page if the session times out. How can I do this?

The easiest way to accomplish this is to set up a redirection in a metatag to occur just after your timeout:

```
<html>
<head>
<META HTTP-EQUIV="REFRESH" CONTENT="910;
 URL=youhavetimedout.csp">
</head>
<body>
<script language="cache" runat="server">
        %session.AppTimeout = 900
</script>
</body>
</html>
```

I want my page to automatically refresh every 60 seconds. How should I do this?

In the head of your CSP page, use the following metatag:

```
META HTTP-EQUIV="REFRESH" CONTENT="60; URL=mypage.csp">
```

Commands and Syntax

How can I display a Caché variable or expression on my CSP page?

A variable or expression can be incorporated into the page at runtime using "#(var)#" or "#(expression)#". For example:

#(name)#, where name has been set

#($G(%request.Get("Username")))#, retrieves Username from the URL

#(2+7+3)#, displays 12 on the Webpage

What is the difference between "#(var)#" and "##(var)##"?

The syntax "#()#" replaces the expression inside the parenthesis with its runtime value. The syntax "##()##" replaces the variable or expression with its value when the page is compiled.

To illustrate the difference, place the following code sample inside a CSP page:

```
Runtime: #($P($H,",",2))#
Compile Time: ##($P($H,",",2))##
```

Open the page in a browser and refresh it a few times. Notice that the *Runtime* value changes each time the page is refreshed. The *Compile Time* value retains the time the page was compiled; it changes only when the page is recompiled.

What is the difference between "#include" and "CSP:Include"?

The **#include** directive allows you to include in your page any text: JavaScript, html, plain text, CSP.

The `<csp:include>` tag includes a properly formatted CSP page; it uses ServerSideRedirect to insert this page and then return to processing the original page.

How can I compile CSP pages?

By default, the browser automatically compiles CSP pages when it loads them, if the pages have changed (based on their timestamp). You can also manually compile your CSP pages in Studio or from the Terminal. In either case, you can control whether to keep the generated source code.

To compile your CSP page using Studio:

1. On the **Tools** menu, click **Options**, and then click the **Compile** tab.
2. Select the **Keep generated source code** check box and click **OK**.
3. Compile your CSP page from the **Build** menu by clicking **Compile**.

To compile your CSP page from the Terminal:

1. From the Terminal, ensure that you are in the correct namespace. If not, change namespaces by entering:

   ```
   zn "<namespace>"
   ```

2. Type: **do $System.CSP.LoadPage("/csp/<namespace>/<pagename>.csp", "ck")**

   For example:

   ```
   SAMPLES> do $System.CSP.LoadPage("/csp/samples/james.csp", "ck")
   ```

Note: The "k" flag tells the compiler to "Keep generated source code."

What are the flags and qualifiers?

For a list of flags, run the following command in the Terminal:

```
Do $System.OBJ.ShowFlags()
```

Note: When using the `f` flag, after the compilation, the existing objects are no longer valid. Use the `v` flag if you want the objects to be valid after class compilation.

For a list of qualifiers, run the following command in the Terminal:

```
Do $System.OBJ.ShowQualifiers()
```

There are a few utility methods I call all the time. How can I avoid using ##class(Package.Class).method()?

Place these methods in a particular class and have your CSP page inherit from that class. Your CSP page can then access these methods by using dot syntax. To do this, use the `<csp:class>` tag as follows:

```
<csp:class super="%CSP.Page,App.Utils">
```

What is a private page?

A private page can only be viewed when called from another CSP page using the proper token. A private page cannot be bookmarked or arrived at by simply typing its URL into the browser. To set a page as private, use the `<csp:class>` tag as follows:

```
<csp:class private=1>
```

I have a set of JavaScript functions and a header that I want on all my pages. How should I include this?

Use the new **#include** syntax:

```
<!--#include file="mystuff.inc"-->
```

This is a textual include, new in Caché 5. The text of the file is automatically included in the page before the page is compiled. Therefore, you can include code that needs to be compiled such as #()# variables, the results of `<csp:query>` queries and `runat="server"` code.

I want to use the <csp:search> tag, but I want to allow the user to search on fields other than ID. Can I do this?

The `<csp:search>` tag has a WHERE attribute which allows you to specify a comma-delimited list of fields on which to search.

```
<csp:search name=mySearch where="Name,Gender" CLASSNAME="Sample.Person">
```

There are several other attributes you can use to customize your <csp:search> functionality. See the <CSP:SEARCH> entry of the *CSP HTML Tag Reference* guide.

Configuration

How do I configure a CSP application to serve pages in a subdirectory?

By using the Management Portal as follows:

1. From the Caché cube, click **Management Portal**. Log in, if necessary.
2. In the Management Portal, navigate to **System Administration** > **Security** > **Applications** > **Web Applications**, click **Edit** for the CSP application to configure.
3. On the **Web Applications** page, click **Edit** for the CSP application to configure.
4. On the **Edit Web Application** page, set **Recurse** to **Yes**.
5. Click **Save**.

I want my users to load my CSP application by pointing their browsers to: http://mydomain.com/banking/login.csp; I do not want /csp/ in the URL. How can I do this?

Use the Management Portal to set up a new CSP application called, for example, /myapp. This process is described in the "Defining a New Application on the CSP Server" section of the *CSP Configuration* chapter of *Using Caché Server Pages (CSP)*.

I have Caché on a different machine than my web server. How can I configure this?

See the "Connecting to Remote Servers" chapter of the *Caché System Administration Guide*.

Miscellaneous

Can I use frames in my CSP application?

Yes. However, you should name the frameset page with a .csp extension. If you create a page called index.html and load CSP pages into the left and right frames, you use two sessions and accordingly two Caché licenses, one for each CSP page. This can cause confusion if you use the session object to store information and you also use unnecessary licenses.

If you call your frameset page index.csp, the result is a single session, which uses one license for that application. Both CSP pages within the frames share this session and any information stored in it.

How do I use a character set with CSP?

What HTTP header information is sent to the browser?

You can view header information in two ways:

- Display your page in the Terminal using the **Show** method:

```
D $System.CSP.Show("/csp/user/mypage.csp")
```

This displays the HTTP headers, as well as the generated HTML source for the page.

- Use the **Head** method of the %Net.HttpRequest class.

```
set http = ##class(%Net.HttpRequest).%New()
set http.server = "localhost"
set http.Port = 57772
do http.Head("csp/samples/loop.csp")
do http.HttpResponse.OutputToDevice()
set http = ""
```

In addition to CSP, I am running Crystal Reports which also uses a .csp extension. How can I make my Caché Server Pages work?

Because CSP and Crystal Reports both use the .csp file extension, there is a conflict if you run both through your web server. Whichever was installed later works, but the earlier application does not. To alleviate this conflict, configure your web server to run one virtual directory for CSP and another for Crystal Reports.

To configure virtual directories using the Internet Services Manager:

1. From the **Start** menu, point to **Settings**, **Control Panel**, **Administrative Tools**, and then click **Internet Services Manager**.
2. Expand the first node, and then expand **Default Web Site**.
3. If CSP was installed last, right-click the Crystal virtual directory and choose **Properties**.

   If Crystal Reports was installed last, right-click the csp virtual directory and choose **Properties**.
4. On the **Virtual Directory** tab of the **Properties** dialog box, click **Configuration** in the lower right portion of the box.
5. Click the **App Mappings** tab and scroll down to find the .csp mapping located near the bottom of this list.
6. If you installed CSP last, change the **Executable Path** for the .csp extension mapping to the location of the Crystal Reports DLL, WSCInSAPI.dll. It is located in the WCS directory of the Crystal install directory. (For example, C:\Program Files\Seagate Software\WCS)

   If you installed Crystal Reports last, change the **Executable Path** for the .csp extension mapping to the location of CSPms.dll, located in the /csp/bin directory of your Caché install directory. (For example, C:\CacheSys\CSP\bin).
7. Click **OK**.