BrowserHawk®
Ensure a Flawless Web Experience

Rules Enforcement Technology (RET)

Users Guide

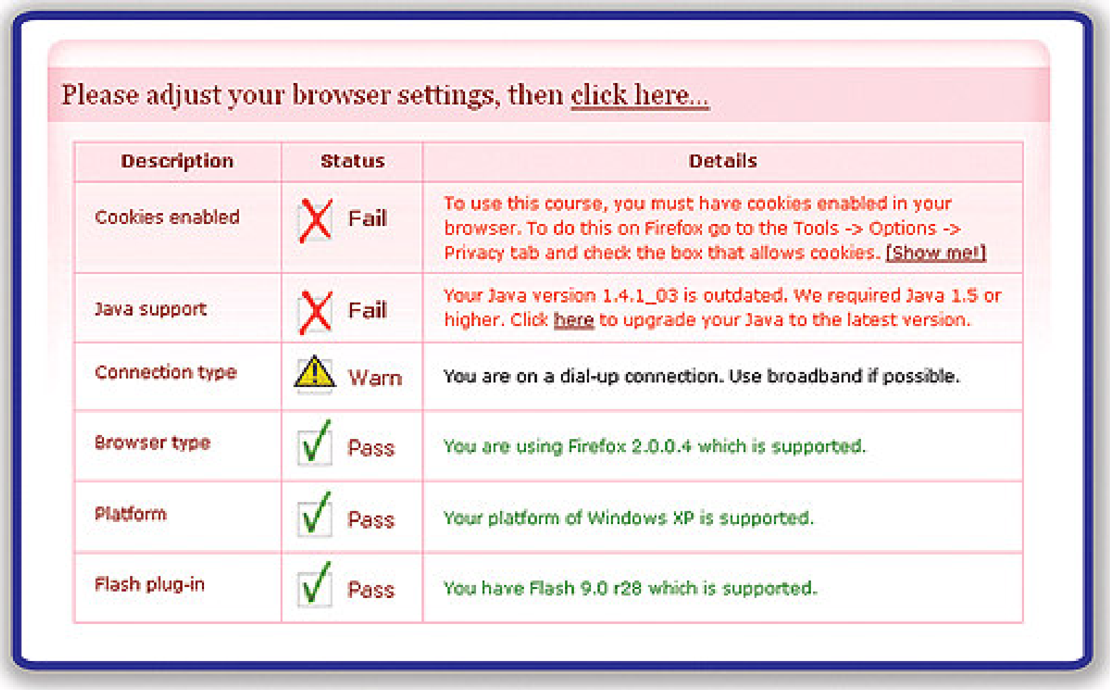
Introduction to RET

BrowserHawk's Rules Enforcement Technology (RET) provides an easy way to ensure that each web site visitor meets your site's minimum browser and system requirements, and to provide automated self-help to those users who do not. And all without having to write any program code!

To use RET you simply define a group of "rules" that reflect your site's requirements. Each rule is configured to evaluate to one of 4 states: Pass, Fail, Warn, and Info. You define the conditions under which these states are assigned. You then pass this rule set to BrowserHawk, which enforces the rules and takes the actions you specify based on the outcomes of its testing.

For example, you may define one rule that a user must have Flash 9 or higher, and that it is failure condition if not. Another rule may be that cookies should be enabled, and it is considered only a warning condition if not.

In one particular RET implementation, as shown below, a "pass/fail" table is presented to the user that summarizes their browser test results. This table contains details about the tests performed, the results, and customized messages such as corrective actions to take. BrowserHawk can fully render this table all on its own, or you can take control and generate the table yourself if you wish.



Please adjust your browser settings, then click here...

Description	Status	Details
Cookies enabled	✗ Fail	To use this course, you must have cookies enabled in your browser. To do this on Firefox go to the Tools -> Options -> Privacy tab and check the box that allows cookies. [Show me!]
Java support	✗ Fail	Your Java version 1.4.1_03 is outdated. We required Java 1.5 or higher. Click here to upgrade your Java to the latest version.
Connection type	⚠ Warn	You are on a dial-up connection. Use broadband if possible.
Browser type	✓ Pass	You are using Firefox 2.0.0.4 which is supported.
Platform	✓ Pass	Your platform of Windows XP is supported.
Flash plug-in	✓ Pass	You have Flash 9.0 r28 which is supported.

In another RET implementation, a user could be redirected to a particular page or site based on the result of any particular rule, rather than shown a results table. Or both styles can be combined, whereby the user is shown the table except if certain conditions are true - in which case the user is redirected instead. Furthermore, for advance usage, you can even programmatically access the test results and write your own custom handlers.

RET provides you with great flexibility and complete customization, including the ability to completely customize the look and feel of the results table using CSS, specify alternative column headings and graphic/icons, and even renaming the states. For example you may prefer "Attention needed" over the default label of "Fail".

RET is also completely extensible. For instance, the logic within the XML rules file can make use of the full set of BrowserHawk standard and extended property values, as well as values from other data sources such as your backend systems. For example, you could add a rule that would display a warning message to the user such as "Your credit card on file expires in 10 days. Please click here to update this now."

Background and Benefits

RET helps you deliver a flawless web experience to every visitor by proactively screening users for unsupported browsers and operating systems, incompatible browser or security settings, missing or outdated browser plug-ins, and much more.

RET significantly cuts the amount of support calls to your help desk and the amount of time spent on each call. Furthermore you will raise the level of satisfaction with your users by enabling most users to correct any issues on-the-spot, without having to first contact support.

The real power of RET is that your browser and system requirements are declared completely within an XML-formatted data source. BrowserHawk then enforces these rules for you automatically, with no custom programming required

Previously, developers would need to write complex, hard-coded "if/then/else" logic to enforce such rules. In addition to this being a time-consuming task, it made maintenance difficult because the code would need to be changed, recompiled and redeployed each time the requirements were updated.

On the contrary, with RET you simply update a centralized XML file or data source. BrowserHawk will then instantly start using your updated rules, without any code changes or updates required.

Rule File Format

The XML rules file consists of a root element <tests> containing a series of <test> elements, each representing a test to apply. Each <test> corresponds to a specific requirement that, if desired, can result in a redirect action or a row in the HTML results table. For example:

```
<tests>
  <!-- Only run on Windows or Mac -->
  <test>...</test>
  <!-- Must be IE 6+ or Firefox 1.5+ -->
  <test>...</test>
</tests>
```

Each <test> element should have a "name" attribute holding a string value for display in the HTML table row. It also should have an "id" attribute as a unique identifier than can be used by CSS for styling this particular test results. To demonstrate:

```
<tests>
  <!-- Only run on Windows or Mac -->
  <test name="Platform Check" id="platform">...</test>
  <!-- Must be IE 6+ or Firefox 1.5+ -->
  <test name="Browser Check" id="browser">...</test>
</tests>
```

Subtest Elements

Within each <test> there may be any number of <pass>, <fail>, <warn>, <info> and <suppress> subtest elements, in any order. Each element contains criteria match logic, and the *first* subtest that matches determines if the test is a pass, a failure, a warning, an info (informational message), or a suppress (resulting in no action or output in the table). For example:

```
<tests>
  <!-- Only run on Windows or Mac -->
  <test name="Platform Check" id="platform">
    <pass>...</pass>
    <warn>...</warn>
    <warn>...</warn>
    <fail>...</fail>
    <info>...</info>
  </test>
  <!-- Must be IE 6+ or Firefox 1.5+ -->
  <test name="Browser Check" id="browser">
    <pass>...</pass>
    <pass>...</pass>
    <fail>...</fail>
    <info>...</info>
    <suppress>...</suppress>
  </test>
</tests>
```

These pass/fail/warn/info/suppress subtests are written using the name of the property under test as the element name, and the match value as the element body. For example, <browser>IE</browser>.

If the value ends with + such as X+ it means X or higher (for numeric types), if it ends in a minus such as X- it means X or lower. An exclamation point !X means not X, an empty element means the empty string (or null value), a * means any value, an X* means starting with X, and false/true match boolean values. True "regular expressions" for advanced matching is also supported (see below).

All match conditions within the subtest must evaluate to true for the subtest to be a match. If there are no match conditions, that by default evaluates to true. All string comparisons are case sensitive. All element names referring to properties must be lower case. If two elements with the same name appear under in the same subtest, that is specially permitted as an indication that either value is acceptable. The below sample demonstrates:

```
<tests>
  <!-- Only run on Windows or Mac -->
  <test name="Platform Check" id="platform">
    <pass>
      <platform>Win*</platform>  <!-- starts with Win -->
      <platform>Mac*</platform>  <!-- starts with Mac -->
    </pass>
    <warn>
      <platform></platform>  <!-- Empty platform indicates unknown -->
    </warn>
    <fail>
      <!-- No test means automatic match if the above don't match -->
    </fail>
  </test>
  <!-- Must be IE 6+ or Firefox 1.5+ -->
  <test name="Browser Check" id="browser">
    <pass>
```

```
      <browser>IE</browser>
      <version>6+</version>
    </pass>
    <pass>
      <browser>Firefox</browser>
      <version>1.5+</version>
    </pass>
    <fail>
      <!-- No test means automatic match if the above don't match -->
    </fail>
  </test>
</tests>
```

Regular Expression Matching

If a subtest property has an XML attribute named "matcher" with the value "regex" then the property value is compared using regular expression matching rules. Regular expressions allow virtually unlimited power in string matching. For more information on regular expression options and behavior, see System.Text.RegularExpressions.Regex (for .NET developers) or java.util.regex.Pattern (for Java developers). Here is an example that uses simple regular expressions to match any Windows or Macintosh platform:

```
<tests>
  <test name="Platform Check" id="platform">
    <pass>
      <platform matcher="regex">(Win|Mac).*</platform>
    </pass>
  </test>
</tests>
```

Message Display

Besides test criteria, each <pass>, <fail>, <warn>, or <info> element may contain within it a <message> to display if this component of the test matches, such as "You have Acrobat 6, however this site requires Acrobat 7." This message will be displayed in the HTML table (or returned as part of a results object if you choose to handle the results programmatically). The value "6" above would be referenced as %plugin_acrobatverex% to enable variable substitution. The special subtest element <suppress> indicates no row should be added to the HTML table as a result of the test. For example:

```
<tests>
  <!-- Only run on Windows or Mac -->
  <test name="Platform Check" id="platform">
    <pass>
      <platform>Win*</platform>  <!-- starts with Win -->
      <platform>Mac*</platform>  <!-- starts with Mac -->
      <message>Your platform %platform% is fine</message>
    </pass>
    <fail>
      <!-- No test means automatic match if the above don't match -->
      <message>Your platform %platform% is not allowed at this time</message>
    </fail>
  </test>
  <!-- Must be IE 6+ or Firefox 1.5+ -->
  <test name="Browser Check" id="browser">
    <pass>
      <browser>IE</browser>
      <version>6+</version>
      <message>You're using IE 6 or later, which is fine</message>
    </pass>
    <pass>
      <browser>Firefox</browser>
      <version>1.5+</version>
      <message>You're using Firefox 1.5 or later, which is fine</message>
    </pass>
    <fail>
      <!-- No test means automatic match if the above doesn't match -->
      <message>%browser% %fullversion% is not supported</message>
    </fail>
  </test>
</tests>
```

Each <pass>, <fail>, <warn>, or <info> element may also, instead of a message, contain a <redirect> to issue upon if this component of the test matches. The element holds the URL to redirect to, which may be relative or absolute.

The URL may also use variable substitution as above, for example to provide a value in a query string. Any redirect action supersedes any HTML table generation. Only the first redirection match is acted upon. As an example, the following issues a redirect it the user is not using IE, Firefox, or Safari:

```
<tests>
  <!-- Redirect if openports doesn't contain 443 or 8080 -->
  <test id="browser" name="BrowserCheck">
    <suppress>
      <browser>IE</browser>
      <browser>Firefox</browser>
      <browser>Safari</browser>
      <!-- No message as this is just a test with a redirect on fail -->
    </suppress>
    <fail>
      <redirect>/update_your_browser.html</redirect>
    </fail>
  </test>
</tests>
```

Both <message> and <redirect> elements may contain variable substitution in their values as a property name surrounded by percent signs: %platform%, %browser%, and so on. To output a percent sign, escape it with a backslash: "You 100\% pass".

In addition to property names, variable substitution keywords can come from the query string (and POST form fields), or cookie values. The search order for a matching keyword value is: BrowserHawk property, JSEval property (discussed later), GET or POST parameter value, then cookie value. This feature allows for external parameterization of the result message or redirect URL. The externally provided values are always encoded to prevent special characters from appearing in the web page or URL. You cannot at this time parameterize a test comparison value, only a message or redirect string.

```
<tests>
  <!-- Embed the user's id taken from the %user% cookie -->
  <test id="size" name="Size">
    <pass>
      <widthavail>500+</widthavail>
      <redirect>/big.aspx?width=%widthavail%&amp;check=true</message>
    </pass>
    <fail>
      <widthavail>500-</widthavail>
      <message>Your browser width is too small.  Please increase your
               browser size, or if you can't please contact technical
               support.  Your user id is %user%.</message>
    </fail>
  </test>
</tests>
```

Notice how the ampersand in the redirect URL was written as &amp; which is the special way to write an ampersand inside an XML document.

Samples and Templates

BrowserHawk ships with a number of samples that demonstrate how to use RET and provide a basic framework for integrating RET into your site. Please refer to those samples which are designed as a learning aid.

In addition, a Visual Studio "New Item Template" is available that automatically creates a default browser help page and all supporting files (graphics, style sheets, default XML rule set), which you can then fully customize. See the BH_RET_Template.zip file in your \Program Files\cyScape\BrowserHawk\dotnet\templates folder for details.

.NET Execution

Applying rules within the .NET environment is a simple multi-step process:

1. Capture the BrowserObj instance as normal.
2. Obtain a RuleEngine instance based on a given XML rule file or string. The instances will be cached internally for efficiency. The method calls are GetFromFile(string filename) and GetFromString(string contents).
3. Obtain an ExtendedOptions instance from the RuleEngine. This options instance will be pre-configured to test everything required by the rules in the XML file.
4. Use the ExtendedOptions instance to get an ExtendedBrowserObj instance.
5. Pass the BrowserObj and ExtendedBrowserObj instances to the GetHtmlResult() method on the RuleEngine. This returns an HTML string holding either the generated table or a redirect command.

```
<%
  BrowserObj brow = BrowserObj.GetBrowser();
  RuleEngine ruleEngine = RuleEngine.GetFromFile("rules.xml");
  ExtendedOptions options = ruleEngine.GetExtendedOptions();
  ExtendedBrowserObj ebrow = BrowserObj.GetExtendedBrowser(options);
  Response.Write(ruleEngine.GetHtmlResult(brow, ebrow));
%>
```

If the rules file is not valid XML, or contains XML that doesn't conform to the required specification, an exception is raised. If the rules file changes on disk, BrowserHawk will automatically load the new version of the file and use it immediately without human intervention.

The rules XML file may over time require different sets of extended properties. So that the program code can adjust automatically without the programmer re-editing the file every time the XML changes, the RuleEngine exposes the GetExtendedOptions() method. It returns an ExtendedOptions instance pre-configured to test everything required by the rules in the XML file.

There is also a GetLogOptions() method returning a LogOptions that can be passed to LogData(). The following example demonstrates how to use these properties to configure ExtendedOptions and LogOptions to execute the proper tests and record the test results in the BrowserHawk Reports Web Service (BRWS):

```
<%
  BrowserObj brow = BrowserObj.GetBrowser();
  RuleEngine ruleEngine = RuleEngine.GetFromFile("rules.xml");
  ExtendedOptions options = ruleEngine.GetExtendedOptions();
  ExtendedBrowserObj ebrow = BrowserObj.GetExtendedBrowser(options);
  Response.Write(ruleEngine.GetHtmlResult(brow, ebrow));
  BrowserObj.LogData(brow, ebrow, ruleEngine.GetLogOptions());
%>
```

Some extended properties, like FontsInstalled and OpenPorts, require extra configuration indicating which fonts and/or ports to test. It's possible to set these properties programmatically on the ExtendedOptions instance returned by GetExtendedOptions():

```
<%
  BrowserObj brow = BrowserObj.GetBrowser();
  RuleEngine ruleEngine = RuleEngine.GetFromFile("rules.xml");
  ExtendedOptions options = ruleEngine.GetExtendedOptions();
  options.PortsToCheck = "80,8000";
  ExtendedBrowserObj ebrow = BrowserObj.GetExtendedBrowser(options);
  Response.Write(ruleEngine.GetHtmlResult(brow, ebrow));
  BrowserObj.LogData(brow, ebrow, ruleEngine.GetLogOptions());
%>
```

This approach is undesirable because it requires the code and rules to be tightly coupled. A better design is to configure these properties in the rules XML file itself. Then the properties can be present already on the returned ExtendedOptions instance. For example to test ports 80 and 8000:

```
<tests>
  <config>
    <extended-options>
      <ports-to-check>80,8000</ports-to-check>
      <page-title>Testing open ports...</page-title>
      <page-message>Please wait.  Testing open ports...</page-message>
    </extended-options>
  </config>
  <test id="ports" name="Port Check">
    <pass>
```

```
      <openports matcher="regex">\b80\b</openports>
      <message>You have port 80 open</message>
    </pass>
    <fail>
      <message>You have port 80 closed</message>
    </fail>
  </test>
</tests>
```

This tests the browser for an open port 80 connection, providing a configured test message and page title during the test. (Note the use of a regex and the \b construct to indicate a "word boundary" on each side of the number to ensure an open port 8000 isn't matched as port 80.)

Many of the ExtendedOptions properties can be set via the <extended-options> block in the rules file: <body-tag>, <page-message>, <page-title>, <plugin-custom-id>, <fonts-to-check>, <ports-to-check>, <high-security-message>, <high-security-link>, <cache-token>, <cache-force-refresh>, <cookie-domain>, <request-type> (legal values are "auto", "cookie", "query-string", and "post"), and <server-url>.

As an alternative to GetHtmlResult(), the RuleEngine class exposes a GetResult() method that provides more control over the execution by returning a RuleEngineResult instance. The RuleEngineResult class exposes the test results programmatically, which and how many tests passed, failed, etc. It also has accessors to retrieve the generated HTML table and to retrieve the redirect URL, allowing full programmatic control over sending the table or redirect. For example, the following determines how many tests failed and warned before taking action:

```
<%
  BrowserObj brow = BrowserObj.GetBrowser();
  RuleEngine ruleEngine = RuleEngine.GetFromFile("rules.xml");
  ExtendedOptions options = ruleEngine.GetExtendedOptions();
  ExtendedBrowserObj ebrow = BrowserObj.GetExtendedBrowser(options);
  RuleEngineResult result = ruleEngine.GetResult(brow, ebrow);
  BrowserObj.LogData(brow, ebrow, ruleEngine.GetLogOptions());

  if (result.FailCount == 0 && result.WarnCount <= 1) {
    // No need to write a summary table or do any redirects
    Response.Write(
     "<p>No HTML table if no failures and no more than one warning...</p>");
    Response.Write("Pass: " + result.PassCount + "<br />");
    Response.Write("Fail: " + result.FailCount + "<br />");
    Response.Write("Warn: " + result.WarnCount + "<br />");
    Response.Write("Info: " + result.InfoCount + "<br />");
  }
  else {
    if (result.Redirect != null) {
      Response.Redirect(result.Redirect);
    }
    else if (result.HtmlTable != null) {
      Response.Write(result.HtmlTable);
    }
  }
%>
```

It's also possible to get the specific ids of all the tests in each category, so that behavior can be customized based on each particular test result:

```
<%
  foreach (string failId in result.FailIds) {
    Response.Write("Failed: " + id + "<br />");
  }
%>
```

And you can examine just a single test id to determine if it passed or failed:

```
<%
  if (result.IsFail("size")) {
    Response.Write("Looks like the size test failed");
  }
%>
```

Finally, there is a GetDataSet() method on RuleEngine that returns a standard System.Data.DataSet object. This returns a .NET DataSet that contains the test results for use with .NET controls that accept DataSet objects, such as Repeater and DataGrid.

BrowserHawk4J (Java) Execution

Please see the .NET section above, as execution and coding of RET under Java is nearly identical to that of .NET.

Using Non-BrowserHawk Properties (API Extensibility)

RET supports the ability to create rules based on data values obtained outside of BrowserHawk. For instance, you could have an ActiveX control that exposes certain values, and RET rules can be based on those values. Likewise rules can be based on data values that come from your backend systems, such as customer information from your database.

To interface RET with external data, you configure a name/value pairing between a property name and a value containing arbitrary JavaScript to execute on the client. This is accomplished on the server-side by using the SetJSEval() method as shown below.

After BrowserHawk's detection completes, the result of executing the arbitrary JavaScript is available to RET for its rules processing. For example, the following returns as a custom property the document title of the test page and displays the user's local time in the results as an informational message:

```
ExtendedOptions options = new ExtendedOptions();
options.SetJSEval("doctitle", "document.title");
options.SetJSEval("localtime", "new Date().toString()");
ExtendedBrowserObj ebrow = BrowserObj.GetExtendedBrowser(options);
string title = ebrow.GetJSEval("doctitle"); // Get the result server-side
string localtime = ebrow.GetJSEval("localtime");
```

The second argument to SetJSEval() can be arbitrarily complex and include any valid JavaScript for execution. Note that if you need additional objects (such as an ActiveX control) or JavaScript functions to be present in order to support your external property, you can add these to the page using the PageMessage parameter of the ExtendedOptions class or via the config section of the XML.

Within the XML rules file, external properties are referenced just as if they were actual BrowserHawk properties (notice how 'localtime' and 'doctitle' were defined above):

```
  <test name="User Time" id="usertime">
    <info>
      <message>Your local time: %localtime%</message>
    </info>
  </test>
  <test name="Page" id="page">
    <warning>
      <doctitle>My Construction Page</doctitle>
      <message>This page is not yet complete!</message>
    </warning>
  </test>
```

An easy way to interface RET with external data from another source, such as your database, is to dynamically write the value into the JavaScript so that RET can easily pick it up from a local JS variable. For instance from the server side you could do:

```
       string billcode = some SQL statement;
       Response.Write("var billcode = ' " + billcode + " '; ";
```

You can then easily pick up this value from RET by adding:

```
       options.SetJSEval("custbillcode", "billcode");
```

Then you simply reference 'custbillcode' within RET to get at the value. For example:

```
  <test name="Billing code" id="bcode">
    <fail>
      <custbillcode>PASTDUE</custbillcode>
      <message>Your account is past due... Cannot continue until paid</message>
    </fail>
    <info>
      <message>Your billing code is %custbillcode%
        and in good standing</message>
    </info>
  </test>
```

For convenience and better design, JSEval properties can be specified directly in the rules XML file that's using them. This allows new JSEval properties to be introduced without having to change your program code. For example:

```
<tests>
  <config>
    <extended-options>
      <jseval name="loc">document.location</jseval>
    </extended-options>
  </config>
  <test id="location" name="Location Check">
    <pass>
      <loc matcher='regex'>https?://localhost.*</loc>
      <message>You're using localhost, good.</message>
    </pass>
    <fail>
```

```
      <message>Sorry, you need to use localhost.</message>
    </fail>
  </test>
</tests>
```

Generated HTML

The generated "pass/fail" HTML table has no special formatting of its own, but exposes class and id attributes which allow for advanced CSS control of the table's appearance. The table itself always has the class="bhawkrules" decoration. The header row has the class="bhawkheader" decoration, with more specific class attributes on each of its <th> elements. Every <tr> row displaying a test that passed has the class="bhawkpass" added to it; every failure has class="bhawkfail"; every warning has class="bhawkwarn"; and every info has class="bhawkinfo". For each of the three columns per row there are more specific class attributes. The following example demonstrates a possible HTML table result as generated by BrowserHawk based on the initial tests we ran:

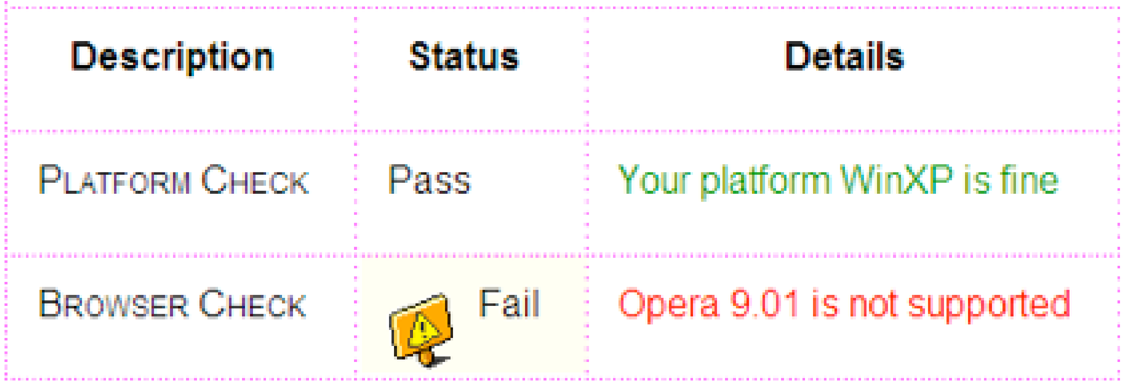
```
<table class="bhawkrules">
  <tr clas="bhawkheader">
    <th class="bhawktestnameheader">Description</th>
    <th class="bhawkteststatusheader">Status</th>
    <th class="bhawktestmessageheader">Details</th>
  </tr>
  <tr class="bhawkpass" id="platform">
    <td class="bhawktestname">Platform Check</td>
    <td class="bhawkteststatus">Pass</td>
    <td class="bhawktestmessage">Your platform Win32 is fine</td>
  </tr>
  <tr class="bhawkfail" id="browser">
    <td class="bhawktestname">Browser Check</td>
    <td class="bhawkteststatus">Fail</td>
    <td class="bhawktestmessage">Opera 9.01 is not supported</td>
  </tr>
</table>
```

A CSS file uses the class and id attributes to modify the display. Below are some CSS rules that could be used to alter the style of the HTML table:

```
.bhawkrules {  /* Formatting of the table */ }
.bhawkheader { /* Formatting for the header row */}
.bhawkpass  {  /* Formatting for all pass rows */ }
.bhawkwarn  {  /* Formatting for all warning rows */ }
.bhawkfail  {  /* Formatting for all failure rows */}
.bhawkinfo  {  /* Formatting for all informational rows */}
.bhawktestname {      /* Formatting for all test name column entries */ }
.bhawkteststatus  {  /* Formatting for all test status column entries */ }
.bhawktestmessage  { /* Formatting for all test message column entries */ }

.bhawkpass .bhawktestmessage {
  /* Formatting for messages when the test passed */
}
.bhawkrules td {
  /* Formatting for all table cells in the rules table */
}
#platform {
  /* Formatting the platform test row */
}
#platform .bhawktestmessage {
  /* Formatting the platform test message cell */
}
.bhawkrules .bhawkpass[id = "browser"] .bhawktestmessage {
   /* Formatting the browser test message cell when it passes */
}
```

As a concrete example, this CSS file makes test names appear in small caps; color codes the pass, fail, warn, and info results; adds a graphic to the browser test result when it fails, and adds a dotted fuchsia line around the table cells:

```
.bhawktestname { font-variant: small-caps; }
.bhawkpass .bhawktestmessage { color: green; }
.bhawkfail .bhawktestmessage { color: red; }
.bhawkwarn .bhawktestmessage { color: yellow; }

.bhawkrules .bhawkfail[id = "browser"] .bhawkteststatus {
  background: #F8F7EF url(http://www.iconarchive.com/icons/iron-devil/ids-3d-icons-20/Board-attention-icon.gif) 10px 1em no-repeat;
  padding-left: 50px;
}

.bhawkrules {
  border-bottom: 1px dotted fuchsia;
  border-left: 1px dotted fuchsia;
}
.bhawkrules td,th {
  padding: 11px 20px 20px 11px;
  border-top: 1px dotted fuchsia;
  border-right: 1px dotted fuchsia;
}
```



Description	Status	Details
PLATFORM CHECK	Pass	Your platform WinXP is fine
BROWSER CHECK	Fail	Opera 9.01 is not supported

Customizing Behaviors

The table's header string values, the table's status column string values, and rules for whether or not to display the table at all can be configured in the rules XML file via a <config> element. This element goes above the first <test>. Example:

```
<tests>
  <config>
    <headers>
      <name>Description Substitute</name>
      <status>Status Substitute</status>
      <message>Details Substitute</message>
    </headers>
    <answers>
      <pass>Yeay!</pass>           <!-- Display instead of "Pass" -->
      <fail>Oh no!!</fail>         <!-- Display instead of "Fail" -->
      <warn>Careful now...</warn>  <!-- Display instead of "Warn" -->
      <info>Did you know?</info>   <!-- Display instead of "Info" -->
    </answers>
    <upon-completion>
      <!-- Here we show the table if there's any "info", "warn", or "fail"
           messages, otherwise redirect the user to "/index.aspx".  The default
           display level is "pass", meaning to show the table always. -->
      <display-level>info</display-level>
      <fallback-redirect>/index.aspx</fallback-redirect>
    </upon-completion>
  </config>
  <test>
    ...
  </test>
</tests>
```

The elements under <headers> override what words are used for the table's header line. To suppress the header line completely, that can be done via CSS:

```
.bhawkheader { display: none; }
```

The elements under <answers> override what words are used for the status column to indicate success or otherwise. To suppress this column completely, that can be done via CSS:

```
.bhawkteststatusheader { display: none; }
.bhawkteststatus { display: none; }
```

The elements under <table-display> dictate when (and if) the HTML table is shown. The <display-level> element dictates the level of result required in order to display the table. For example, <display-level>info</display-level> indicates to show it for "info" or anything more serious. If all the results are "pass" then the table is not shown and instead the user is redirected to the location given by <fallback-redirect>. The redirect string may be parameterized with variable substitution (i.e. <fallback-redirect>http://www.someplace.com?loc=%SomeQueryStringParam%></fallback-redirect>.

If any <display-level> is given, then the <fallback-redirect> must also be given. The default <display-level> is "pass" meaning to show the table always. (Unless technically all results are suppressed in which case there will be a table with a header line but no rows.)

Note the <headers>, <answers>, and <table-display> holder elements are optional and may be ignored but are recommended for readability.

Example XML Data to Enforce Rules

```
<tests>
```

```
<config>
  <extended-options>
    <page-title>Testing - please wait</page-title>
    <page-message>Testing, please wait a sec...</</page-message>
      <body-tag>BGCOLOR="#FFFFFF"</body-tag>
  </extended-options>
</config>

<test name="Platform" id="platform">
  <pass>
    <platform>Win*</platform>
    <message>Your platform of %platform% is fine</message>
  </pass>
  <warn>
    <platform>Mac*</platform>
    <message>You may have problems with your Mac browser</message>
  </warn>
  <fail>
    <message>Your platform of %platform% is not supported</message>
  </fail>
</test>

<test name="Browser" id="browser">
  <pass>
    <browser>IE</browser>
    <majorver>6</majorver>
    <message>IE 6 is getting old, but OK</message>
  </pass>
  <pass>
    <browser>IE</browser>
    <majorver>7+</majorver>
    <message>IE %version% is OK</message>
  </pass>
  <pass>
    <browser>Firefox/browser>
    <majorver>2+</majorver>
    <message>Firefox %version% is good</message>
  </pass>
  <fail>
    <message>You must have IE 6 or higher, or Firefox</message>
  </fail>
</test>

<test name="Acrobat" id="acrobat">
  <warn>
    <plugin_acrobatverex>8+</plugin_acrobatverex>
    <message>Our PDFs haven't been tested with Acrobat 8+ yet</message>
  </warn>
  <pass>
    <plugin_acrobatverex>6.0.1+</plugin_acrobatverex>
    <message>Adobe Acrobat Reader %plugin_acrobatverex% is OK</message>
  </pass>
  <fail>
    <message>Adobe Acrobat Reader before 6.0.1 not supported</message>
  </fail>
</test>

<test name="Monitor" id="monitor">
  <pass>
    <colordepth>16+</colordepth>
    <width>1024+</width>
    <height>768+</height>
    <message>Your monitor passes inspection</message>
    </pass>
    <fail>
      <colordepth>15-</colordepth>
      <message>Your colordepth %colordepth%-bit is too low</message>
    </fail>
    <fail>
      <width>1023-</width>
      <height>767-</height>
      <message>Your resolution %width%x%height% is too low</message>
    </fail>
  </test>

  <test name="Connection type" id="connection">
    <pass>
      <broadband>true</broadband>
      <message>Glad to see you are using broadband</message>
    </pass>
    <fail>
      <message>We don't support use on dial-up, sorry</message>
    </fail>
  </test>
</tests>
```