Available Checks

ht version v1.0.0

Body

```
type Body Condition
Body provides simple condition checks on the response body.
```

DeleteCookie

```
type DeleteCookie struct {
    Name   string
    Path   string `json:",omitempty"`
    Domain string `json:",omitempty"`
}
```

```
DeleteCookie checks that the HTTP response properly deletes all cookies
matching Name, Path and Domain. Path and Domain are optional in which case
all cookies with the given Name are checked for deletion.
```

FinalURL

```
type FinalURL Condition
FinalURL checks the last URL after following all redirects. This check is
useful only for tests with Request.FollowRedirects=true
```

HTMLContains

```
type HTMLContains struct {
    // Selector is the CSS selector of the HTML elements.
    Selector string

    // Text contains the expected plain text content of the HTML elements
    // selected through the given selector.
    Text []string `json:",omitempty"`

    // Raw turns of white space normalization and will check the unprocessed
    // text content.
    Raw bool `json:",omitempty"`

    // Complete makes sure that no excess HTML elements are found:
    // If true the len(Text) must be equal to the number of HTML elements
    // selected for the check to succeed.
    Complete bool `json:",omitempty"`

    // InOrder makes the check fail if the selected HTML elements have a
    // different order than given in Text.
    InOrder bool `json:",omitempty"`

    // Has unexported fields.
}
```

```
HTMLContains checks the text content (and optionally the order) of HTML
elements selected by a CSS rule.

The text content found in the HTML document is normalized by roughly the
following procedure:
```

```
1.  Newlines are inserted around HTML block elements
    (i.e. any non-inline element)
2.  Newlines and tabs are replaced by spaces.
3.  Multiple spaces are replaced by one space.
4.  Leading and trailing spaces are trimmed of.

As an example consider the following HTML:

<html><body>
  <ul class="fancy"><li>One</li><li>S<strong>econ</strong>d</li><li> Three </li></ul>
</body></html>

The normalized text selected by a Selector of "ul.fancy" would be

"One Second Three"
```

HTMLTag

```
type HTMLTag struct {
    // Selector is the CSS selector of the HTML elements.
    Selector string

    // Count determines the number of occurrences to check for:
    //     < 0: no occurrence
    //    == 0: one ore more occurrences
    //     > 0: exactly that many occurrences
    Count int `json:",omitempty"`

    // Has unexported fields.
}
```

```
HTMLTag checks for the existens of HTML elements selected by CSS selectors.
```

Header

```
type Header struct {
    // Header is the HTTP header to check.
    Header string

    // Condition is applied to the first header value. A zero value checks
    // for the existence of the given Header only.
    Condition `json:",omitempty"`

    // Absent indicates that no header Header shall be part of the response.
    Absent bool `json:",omitempty"`
}
```

```
Header provides a textual test of single-valued HTTP headers.
```

JSON

```
type JSON struct {
    // Element in the flattened JSON map to apply the Condition to.
    // E.g.  "foo.2" in "{foo: [4,5,6,7]}" would be 6.
    // An empty value result in just a check for 'wellformedness' of
    // the JSON.
    Element string `json:",omitempty"`

    // Condition to apply to the value selected by Element.
    // If Condition is the zero value then only the existence of
    // a JSON element selected by Element is checked.
    // Note that Condition is checked against the actual value in the
    // flattened JSON map which will contain the quotation marks for
    // string values.
    Condition `json:",omitempty"`

    // Sep is the separator in Element when checking the Condition.
    // A zero value is equivalent to "."
    Sep string `json:",omitempty"`
}
```

JSON allow to check a single string, number, boolean or null element in a JSON document against a Condition.

Elements of the JSON document are selected by an element selector. In the JSON document

```
{ "foo": 5, "bar": [ 1, "qux", 3 ], "waz": true, "nil": null }
```

the follwing element selector are present and have the shown values:

```
foo       5
bar.0     1
bar.1     "qux"
bar.2     3
waz       true
nil       null
```

JSONExpr

```
type JSONExpr struct {
    // Expression is a boolean gojee expression which must evaluate
    // to true for the check to pass.
    Expression string `json:",omitempty"`

    // Has unexported fields.
}
```

JSONExpr allows checking JSON documents via gojee expressions. See github.com/nytlabs/gojee (or the vendored version) for details.

Consider this JSON:

```
{ "foo": 5, "bar": [ 1, 2, 3 ] }
```

The follwing expression have these truth values:

```
.foo == 5                     true
$len(.bar) > 2                true as $len(.bar)==3
.bar[1] == 2                  true
(.foo == 9) || (.bar[0]<7)    true as .bar[0]==1
$max(.bar) == 3               true
$has(.bar, 7)                 false as bar has no 7
```

Latency

```
type Latency struct {
    // N is the number if request to measure. It should be much larger
    // than Concurrent. Default is 50.
    N int `json:",omitempty"`

    // Concurrent is the number of concurrent requests in flight.
    // Defaults to 2.
    Concurrent int `json:",omitempty"`

    // Limits is a string of the following form:
```

```
    //    "50% ≤ 150ms; 80% ≤ 200ms; 95% ≤ 250ms; 0.9995 ≤ 0.9s"
    // The limits above would require the median of the response
    // times to be <= 150 ms and would allow only 1 request in 2000 to
    // exced 900ms.
    // Note that it must be the ≤ sign (U+2264), a plain < or a <=
    // is not recognized.
    Limits string `json:",omitempty"`

    // IndividualSessions tries to run the concurrent requests in
    // individual sessions: A new one for each of the Concurrent many
    // requests (not N many sessions).
    // This is done by using a fresh cookiejar so it won't work if the
    // request requires prior login.
    IndividualSessions bool `json:",omitempty"`

    // If SkipChecks is true no checks are performed i.e. only the
    // requests are executed.
    SkipChecks bool `json:",omitempty"`

    // DumpTo is the filename where the latencies are reported.
    // The special values "stdout" and "stderr" are recognized.
    DumpTo string `json:",omitempty"`

    // Has unexported fields.
}
```

Latency provides checks against percentils of the response time latency.

Links

```
type Links struct {
    // Which links to test; a combination of "a", "img", "link" and "script".
    // E.g. use "a img" to check the href of all a tags and src of all img tags.
    Which string

    // Head triggers HEAD requests instead of GET requests.
    Head bool `json:",omitempty"`

    // Concurrency determines how many of the found links are checked
    // concurrently. A zero value indicates sequential checking.
    Concurrency int `json:",omitempty"`

    // Timeout is the client timeout if different from main test.
    Timeout Duration `json:",omitempty"`

    // OnlyLinks and IgnoredLinks can be used to select only a subset of
    // all links.
    OnlyLinks, IgnoredLinks []Condition `json:",omitempty"`

    // Has unexported fields.
}
```

Links checks links and references in HTML pages for availability.

Logfile

```
type Logfile struct {
    // Path is the file system path to the logfile."
    Path string

    // Condition the written stuff must fulfill.
    Condition `json:",omitempty"`

    // Disallow states what is forbidden in the written log.
    Disallow []string `json:",omitempty"`

    // Has unexported fields.
}
```

Logfile provides checks on files (i.e. it ignores the response). During preparation the current file size is determined and the checks are run against the bytes written after preparation.

SetCookie

```
type SetCookie struct {
    Name   string    `json:",omitempty"` // Name is the cookie name.
    Value  Condition `json:",omitempty"` // Value is applied to the cookie value
    Path   Condition `json:",omitempty"` // Path is applied to the path value
    Domain Condition `json:",omitempty"` // Domain is applied to the domain value

    // MinLifetime is the expectetd minimum lifetime of the cookie.
    // A positive value enforces a persistent cookie.
    // Negative values are illegal (use DelteCookie instead).
    MinLifetime Duration `json:",omitempty"`

    // Absent indicates that the cookie with the given Name must not be received.
    Absent bool `json:",omitempty"`

    // Type is the type of the cookie. It is a space separated string of
    // the following (case-insensitive) keywords:
    //   - "session": a session cookie
    //   - "persistent": a persistent cookie
    //   - "secure": a secure cookie, to be sont over https only
    //   - "unsafe", aka insecure; to be sent also over http
    //   - "httpOnly": not accesible from JavaScript
    //   - "exposed": accesible from JavaScript, Flash, etc.
    Type string `json:",omitempty"`
}
```

SetCookie checks for cookies being properly set. Note that the Path and Domain conditions are checked on the received Path and/or Domain and not on the interpreted values according to RFC 6265.

Sorted

```
type Sorted struct {
    // Text is the list of text fragments to look for in the
    // response body or the normalized text content of the
    // HTML page.
    Text []string

    // AllowedMisses is the number of elements of Text which may
    // not be present in the response body. The default of 0 means
    // all elements of Text must be present.
    AllowedMisses int `json:",omitempty"`
}
```

Sorted checks for an ordered occurrence of items. The check Sorted could be replaced by a Regexp based Body test without loss of functionality; Sorted just makes the idea of "looking for a sorted occurrence" clearer.

If the response has a Content-Type header indicating a HTML response the HTML will be parsed and the text content normalized as described in the HTMLContains check.

XML

```
type XML struct {
    // Path is a XPath expression understood by launchpad.net/xmlpath.
    Path string

    // Condition the first element addressed by Path must fulfill.
    Condition

    // Has unexported fields.
}
```

XML allows to check XML request bodies.

AnyOne

```
type AnyOne struct {
    // Of is the list of checks to execute.
    Of CheckList
}
```

AnyOne checks that at least one Of the embedded checks passes. It is the short circuiting boolean OR of the underlying checks. Check execution stops once the first passing check is found. Example (in JSON5 notation) to check status code for '202 OR 404':

```
{
    Check: "AnyOne", Of: [
        {Check: "StatusCode", Expect: 202},
        {Check: "StatusCode", Expect: 404},
    ]
}
```

ContentType

```
type ContentType struct {
    // Is is the wanted content type. It may be abrevated, e.g.
    // "json" would match "application/json"
    Is string

    // Charset is an optional charset
    Charset string `json:",omitempty"`
}
```

ContentType checks the Content-Type header.

Identity

```
type Identity struct {
    // SHA1 is the expected hash as shown by sha1sum of the whole body.
    // E.g. 2ef7bde608ce5404e97d5f042f95f89f1c232871 for a "Hello World!"
    // body (no newline).
    SHA1 string
}
```

Identity checks the value of the response body by comparing its SHA1 hash to the expected SHA1 value.

Image

```
type Image struct {
    // Format is the format of the image as registered in package image.
    Format string `json:",omitempty"`

    // If > 0 check width or height of image.
    Width, Height int `json:",omitempty"`

    // Fingerprint is either the 16 hex digit long Block Mean Value hash or
    // the 24 hex digit long Color Histogram hash of the image.
    Fingerprint string `json:",omitempty"`

    // Threshold is the limit up to which the received image may differ
    // from the given BMV or ColorHist fingerprint.
    Threshold float64 `json:",omitempty"`
}
```

Image checks image format, size and fingerprint. As usual a zero value of a field skips the check of that property. Image fingerprinting is done via github.com/vdobler/ht/fingerprint. Only one of BMV or ColorHist should be used as there is just one threshold.

NoServerError

```
type NoServerError struct{}
```

NoServerError checks the HTTP status code for not being a 5xx server error and that the body could be read without errors or timeouts.

None

```
type None struct {
    // Of is the list of checks to execute.
    Of CheckList
}
```

None checks that none Of the embedded checks passes. It is the NOT of the short circuiting boolean AND of the underlying checks. Check execution stops once the first passing check is found. It Example (in JSON5 notation) to check for non-occurrence of 'foo' in body:

```
{
    Check: "None", Of: [
        {Check: "Body", Contains: "foo"},
    ]
}
```

Redirect

```
type Redirect struct {
    // To is matched against the Location header. It may begin with,
    // end with or contain three dots "..." which indicate that To should
    // match the end, the start or both ends of the Location header
    // value. (Note that only one occurrence of "..." is supported."
    To string

    // If StatusCode is greater zero it is the required HTTP status code
    // expected in this response. If zero, the valid status codes are
    // 301 (Moved Permanently), 302 (Found), 303 (See Other) and
    // 307 (Temporary Redirect)
    StatusCode int `json:",omitempty"`
}
```

Redirect checks for a singe HTTP redirection.

Note that this check cannot be used on tests with

```
Request.FollowRedirects = true
```

as Redirect checks only the final response which will not be a redirection if redirections are followed automatically.

RedirectChain

```
type RedirectChain struct {
    // Via contains the necessary URLs accessed during a redirect chain.
    //
    // Any URL may start with, end with or contain three dots "..." which
    // indicate a suffix, prefix or suffix+prefix match like in the To
    // field of Redirect.
    Via []string
}
```

RedirectChain checks steps in a redirect chain. The check passes if all stations in Via have been accessed in order; the actual redirect chain may hit additional stations.

Note that this check can be used on tests with

```
Request.FollowRedirects = true
```

Resilience

```
type Resilience struct {
    // Methods is the space separated list of HTTP methods to check,
    // e.g. "GET POST HEAD". The empty value will test the original
    // method of the test.
    Methods string `json:",omitempty"`

    // ModParam and ModHeader control which modifications of parameter values
    // and header values are checked.
    // It is a space separated string of the modifications explained above
    // e.g. "drop nonsense empty".
    // An empty value turns off resilience testing.
    ModParam, ModHeader string `json:",omitempty"`

    // ParamsAs controls how parameter values are transmitted, it
    // is a space separated list of all transmission types like in
    // the Request.ParamsAs field, e.g. "URL body multipart" to check
    // URL query parameters, x-www-form-urlencoded and multipart/formdata.
    // The empty value will just check the type used in the original
    // test.
    ParamsAs string `json:",omitempty"`

    // SaveFailuresTo is the filename to which all failed checks shall
    // be logged. The data is appended to the file.
    SaveFailuresTo string `json:",omitempty"`

    // Checks is the list of checks to perform on the received responses.
    // In most cases the -- correct -- behaviour of the server will differ
    // from the response to a valid, unscrambled request; typically by
    // returning one of the 4xx status codes.
    // If Checks is empty, only a simple NoServerError will be executed.
    Checks CheckList `json:",omitempty"`
    // Values contains a list of values to use as header and parmater values.
    // Note that header and parameter checking uses the same list of Values,
    // you might want to do two Resilience checks, one for the headers and one
    // for the parameters.
    // If values is empty, then only the builtin modifications selected by
    // Mod{Param,Header} are used.
    Values []string
}
```

```
Resilience checks the resilience of an URL against unexpected requests like
different HTTP methods, changed or garbled parameters, different parameter
transmission types and changed or garbled HTTP headers.

Parameters and Header values can undergo several different types of
modifications

* all:        all the individual modifications below
* drop:       don't send at all
* none:       don't modify the individual parameters or header but
              don't send any parameters or headers
* double:     send same value two times
* twice:      send two different values (original and "extraValue")
* change:     change a single character (first, middle and last one)
* delete:     drop single character (first, middle and last one)
* nonsense:   the values "p,f1u;p5c:h*", "hubba%12bubba(!" and "\t \v \r \n "
* malicious: the values "\x00\x00", "\uFEFF\u200B\u2029", "ʇunpᴉpᴉɔuᴉ",
              "http://a/%%30%30" and "' OR 1=1 -- 1"
* user        use user defined values from Values
* empty:      ""
* type:       change the type (if obvious)
    - "1234"     -->  "wwww"
    - "3.1415"   -->  "wwwwww"
    - "i@you.me" -->  "iXyouYme"
    - "foobar  " -->  "123"
* large:      produce much larger values
    - "1234"     -->  "9999999" (just large), "2147483648" (MaxInt32 + 1)
                      "9223372036854775808" (MaxInt64 + 1)
                      "18446744073709551616" (MaxUInt64 + 1)
    - "56.78"    -->  "888888888.9999", "123.456e12",
                      "3.5e38" (larger than MaxFloat32)
                      "1.9e308" (larger than MaxFloat64)
    - "foo"      -->  50 * "X", 160 * "Y" and 270 * "Z"
* tiny:       produce 0 or short values
    - "1234"      -->  "0" and "1"
    - "12.3"      -->  "0", "0.02", "0.0003", "1e-12" and "4.7e-324"
    - "foobar"    --> "f"
* negative    produce negative values
    - "1234"      -->  "-2"
    - "56.78"     -->  "-3.3"

This check will make a wast amount of request to the given URL including the
modifying and non-idempotent methods POST, PUT, and DELETE. Some care using
this check is advisable.
```

ResponseTime

```
type ResponseTime struct {
    Lower  Duration `json:",omitempty"`
    Higher Duration `json:",omitempty"`
}
```

```
ResponseTime checks the response time.
```

StatusCode

```
type StatusCode struct {
    // Expect is the value to expect, e.g. 302.
    //
    // If Expect <= 9 it matches a whole range of status codes, e.g.
    // with Expect==4 any of the 4xx status codes would fulfill this check.
    Expect int
}
```

StatusCode checks the HTTP statuscode.

UTF8Encoded

```
type UTF8Encoded struct{}
```

UTF8Encoded checks that the response body is valid UTF-8 without BOMs.

ValidHTML

```
type ValidHTML struct {
    // Ignore is a space separated list of issues to ignore.
    // You normaly won't skip detection of these issues as all issues
    // are fundamental flaw which are easy to fix.
    Ignore string `json:",omitempty"`
}
```

ValidHTML checks for valid HTML 5; well kinda: It make sure that some common but easy to detect fuckups are not present. The following issues are detected:

```
* 'doctype':   not exactly one DOCTYPE
* 'structure': ill-formed tag nesting / tag closing
* 'uniqueids': uniqness of id attribute values
* 'lang':      ill-formed lang attributes
* 'attr':      dupplicate attributes
* 'escaping':  unescaped &, < and > characters or unknown entities
* 'label':     reference to nonexisting id in label tags
* 'url':       malformed URLs
```

Notes:

- HTML5 allows unescaped & in several circumstances but ValidHTML reports all stray & as an error.
- The lang attributes are parse very lax, e.g. the non-canonical form 'de_CH' is considered valid (and equivalent to 'de-CH'). I don't know how browser handle this.

W3CValidHTML

```
type W3CValidHTML struct {
    // AllowedErrors is the number of allowed errors (after ignoring errors).
    AllowedErrors int `json:",omitempty"`

    // IgnoredErrros is a list of error messages to be ignored completely.
    IgnoredErrors []Condition `json:",omitempty"`
}
```

W3CValidHTML checks for valid HTML but checking the response body via the online checker from W3C which is very strict.

Type Condition is not a Check but it is used so often in checks that it is worth describing here.

Condition

```
type Condition struct {
    // Equals is the exact value to be expected.
    // No other tests are performed if Equals is non-zero as these
    // other tests would be redundant.
    Equals string `json:",omitempty"`

    // Prefix is the required prefix
    Prefix string `json:",omitempty"`

    // Suffix is the required suffix.
    Suffix string `json:",omitempty"`

    // Contains must be contained in the string.
    Contains string `json:",omitempty"`

    // Regexp is a regular expression to look for.
    Regexp string `json:",omitempty"`

    // Count determines how many occurrences of Contains or Regexp
    // are required for a match:
    //     0: Any positive number of matches is okay
    //   > 0: Exactly that many matches required
    //   < 0: No match allowed (invert the condition)
    Count int `json:",omitempty"`

    // Min and Max are the minimum and maximum length the string may
    // have. Two zero values disables this test.
    Min, Max int `json:",omitempty"`

    // Has unexported fields.
}
```

```
Condition is a conjunction of tests against a string. Note that Contains and
Regexp conditions both use the same Count; most likely one would use either
Contains or Regexp but not both.
```