html / template

Package template go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Doc Overview Subdirectories Versions Imports Imported By Licenses

Overview

Package template (html/template) implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package text/template and should be used instead of text/template whenever the output is HTML.

The documentation here focuses on the security features of the package. For information about how to program the templates themselves, see the documentation for text/template.

Introduction

This package wraps package text/template so you can share its template API to parse and execute HTML templates safely.

```
tmpl, err := template.New("name").Parse(...)
// Error checking elided
err = tmpl.Execute(out, data)
```

If successful, tmpl will now be injection-safe. Otherwise, err is an error defined in the docs for ErrorCode.

HTML templates treat data values as plain text which should be encoded so they can be safely embedded in an HTML document. The escaping is contextual, so actions can appear within JavaScript, CSS, and URI contexts.

The security model used by this package assumes that template authors are trusted, while Execute's data parameter is not. More details are provided below.

Example

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces

```
Hello, <script>alert('you have been pwned')</script>!
```

but the contextual autoescaping in html/template

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces safe, escaped HTML output

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/script&gt;!
```

Contexts

This package understands HTML, CSS, JavaScript, and URIs. It adds sanitizing functions to each simple action pipeline, so given the excerpt

```
<a href="/search?q={{.}}">{{.}}</a>
```

At parse time each {{.}} is overwritten to add escaping functions as necessary. In this case it becomes

```
<a href="/search?q={{. | urlescaper | attrescaper}}">{{. | htmlescaper}}</a>
```

where urlescaper, attrescaper, and htmlescaper are aliases for internal escaping functions.

For these internal escaping functions, if an action pipeline evaluates to a nil interface value, it is treated as though it were an empty string.

Namespaced and data- attributes

Attributes with a namespace are treated as if they had no namespace. Given the excerpt

```
<a my:href="{{.}}"></a>
```

At parse time the attribute will be treated as if it were just "href". So at parse time the template becomes:

```
<a my:href="{{. | urlescaper | attrescaper}}"></a>
```

Similarly to attributes with namespaces, attributes with a "data-" prefix are treated as if they had no "data-" prefix. So given

```
<a data-href="{{.}}"></a>
```

At parse time this becomes

```
<a data-href="{{. | urlescaper | attrescaper}}"></a>
```

If an attribute has both a namespace and a "data-" prefix, only the namespace will be removed when determining the context. For example

```
<a my:data-href="{{.}}"></a>
```

This is handled as if "my:data-href" was just "data-href" and not "href" as it would be if the "data-" prefix were to be ignored too. Thus at parse time this becomes just

```
<a my:data-href="{{. | attrescaper}}"></a>
```

As a special case, attributes with the namespace "xmlns" are always treated as containing URLs. Given the excerpts

```
<a xmlns:title="{{.}}"></a>
<a xmlns:href="{{.}}"></a>
<a xmlns:onclick="{{.}}"></a>
```

At parse time they become:

```
<a xmlns:title="{{. | urlescaper | attrescaper}}"></a>
<a xmlns:href="{{. | urlescaper | attrescaper}}"></a>
<a xmlns:onclick="{{. | urlescaper | attrescaper}}"></a>
```

Errors

See the documentation of ErrorCode for details.

A fuller picture

The rest of this package comment may be skipped on first reading; it includes details necessary to understand escaping contexts and error messages. Most users will not need to understand these details.

Contexts

Assuming {{.}} is `O'Reilly: How are <i>you</i>?`, the table below shows how {{.}} appears when used in the context to the left.

```
Context                          {{.}} After
{{.}}                            O'Reilly: How are &lt;i&gt;you&lt;/i&gt;?
<a title='{{.}}'>                O&#39;Reilly: How are you?
<a href="/{{.}}">                O&#39;Reilly: How are %3ci%3eyou%3c/i%3e?
<a href="?q={{.}}">              O&#39;Reilly%3a%20How%20are%3ci%3e...%3f
```

```
<a onx='f("{{.}}")'>              O\x27Reilly: How are \x3ci\x3eyou...?
<a onx='f({{.}})'>                "O\x27Reilly: How are \x3ci\x3eyou...?"
<a onx='pattern = /{{.}}/;'>      O\x27Reilly: How are \x3ci\x3eyou...\x3f
```

If used in an unsafe context, then the value might be filtered out:

```
Context                           {{.}} After
<a href="{{.}}">                  #ZgotmplZ
```

since "O'Reilly:" is not an allowed protocol like "http:".

If {{.}} is the innocuous word, `left`, then it can appear more widely,

```
Context                              {{.}} After
{{.}}                                left
<a title='{{.}}'>                    left
<a href='{{.}}'>                     left
<a href='/{{.}}'>                    left
<a href='?dir={{.}}'>                left
<a style="border-{{.}}: 4px">        left
<a style="align: {{.}}">             left
<a style="background: '{{.}}'>       left
<a style="background: url('{{.}}')>  left
<style>p.{{.}} {color:red}</style>   left
```

Non-string values can be used in JavaScript contexts. If {{.}} is

```
struct{A,B string}{ "foo", "bar" }
```

in the escaped template

```
<script>var pair = {{.}};</script>
```

then the template output is

```
<script>var pair = {"A": "foo", "B": "bar"};</script>
```

See package json to understand how non-string content is marshaled for embedding in JavaScript contexts.

Typed Strings

By default, this package assumes that all pipelines produce a plain text string. It adds escaping pipeline stages necessary to correctly and safely embed that plain text string in the appropriate context.

When a data value is not plain text, you can make sure it is not over-escaped by marking it with its type.

Types HTML, JS, URL, and others from content.go can carry safe content that is exempted from escaping.

The template

```
Hello, {{.}}!
```

can be invoked with

```
tmpl.Execute(out, template.HTML(`<b>World</b>`))
```

to produce

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

instead of the

```
Hello, &lt;b&gt;World&lt;b&gt;!
```

that would have been produced if {{.}} was a regular string.

Security Model

https://rawgit.com/mikesamuel/sanitized-jquery-templates/trunk/safetemplate.html#problem_definition defines "safe" as used by this package.

This package assumes that template authors are trusted, that Execute's data parameter is not, and seeks to preserve the properties below in the face of untrusted data:

Structure Preservation Property: "... when a template author writes an HTML tag in a safe templating language, the browser will interpret the corresponding portion of the output as a tag regardless of the values of untrusted data, and similarly for other structures such as attribute boundaries and JS and CSS string boundaries."

Code Effect Property: "... only code specified by the template author should run as a result of injecting the template output into a page and all code specified by the template author should run as a result of the same."

Least Surprise Property: "A developer (or code reviewer) familiar with HTML, CSS, and JavaScript, who knows that contextual autoescaping happens should be able to look at a {{.}} and correctly infer what sanitization happens."

func HTMLEscape

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

func HTMLEscapeString

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

func HTMLEscaper

```
func HTMLEscaper(args ...interface{}) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

func IsTrue

```
func IsTrue(val interface{}) (truth, ok bool)
```

IsTrue reports whether the value is 'true', in the sense of not the zero of its type, and whether the value has a meaningful truth value. This is the definition of truth used by if and other such actions.

func JSEscape

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

func JSEscapeString

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

func JSEscaper

```
func JSEscaper(args ...interface{}) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

func URLQueryEscaper

```
func URLQueryEscaper(args ...interface{}) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

type CSS

```
type CSS string
```

CSS encapsulates known safe content that matches any of:

```
1. The CSS3 stylesheet production, such as `p { color: purple }`.
2. The CSS3 rule production, such as `a[href=~"https:"].foo#bar`.
3. CSS3 declaration productions, such as `color: red; margin: 2px`.
4. The CSS3 value production, such as `rgba(0, 0, 255, 127)`.
```

See https://www.w3.org/TR/css3-syntax/#parsing and https://web.archive.org/web/20090211114933/http://w3.org/TR/css3-syntax#style

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type Error

```
type Error struct {
    // ErrorCode describes the kind of error.
    ErrorCode ErrorCode
    // Node is the node that caused the problem, if known.
    // If not nil, it overrides Name and Line.
    Node parse.Node
    // Name is the name of the template in which the error was encountered.
    Name string
    // Line is the line number of the error in the template source or 0.
    Line int
    // Description is a human-readable description of the problem.
    Description string
}
```

Error describes a problem encountered during template Escaping.

func (*Error) Error

```
func (e *Error) Error() string
```

type ErrorCode

```
type ErrorCode int
```

ErrorCode is a code for a kind of error.

```
const (
    // OK indicates the lack of an error.
    OK ErrorCode = iota

    // ErrAmbigContext: "... appears in an ambiguous context within a URL"
    // Example:
    //   <a href="
    //      {{if .C}}
    //        /path/
    //      {{else}}
    //        /search?q=
    //      {{end}}
    //      {{.X}}
    //   ">
    // Discussion:
    //   {{.X}} is in an ambiguous URL context since, depending on {{.C}},
    //  it may be either a URL suffix or a query parameter.
    //   Moving {{.X}} into the condition removes the ambiguity:
    //   <a href="{{if .C}}/path/{{.X}}{{else}}/search?q={{.X}}">
    ErrAmbigContext

    // ErrBadHTML: "expected space, attr name, or end of tag, but got ...",
    //   "... in unquoted attr", "... in attribute name"
    // Example:
    //   <a href = /search?q=foo>
    //   <href=foo>
    //   <form na<e=...>
    //   <option selected<
    // Discussion:
    //   This is often due to a typo in an HTML element, but some runes
    //   are banned in tag names, attribute names, and unquoted attribute
    //   values because they can tickle parser ambiguities.
    //   Quoting all attributes is the best policy.
    ErrBadHTML

    // ErrBranchEnd: "{{if}} branches end in different contexts"
    // Example:
    //   {{if .C}}<a href="{{end}}{{.X}}
    // Discussion:
    //   Package html/template statically examines each path through an
    //   {{if}}, {{range}}, or {{with}} to escape any following pipelines.
    //   The example is ambiguous since {{.X}} might be an HTML text node,
    //   or a URL prefix in an HTML attribute. The context of {{.X}} is
    //   used to figure out how to escape it, but that context depends on
    //   the run-time value of {{.C}} which is not statically known.
    //
    //   The problem is usually something like missing quotes or angle
    //   brackets, or can be avoided by refactoring to put the two contexts
    //   into different branches of an if, range or with. If the problem
    //   is in a {{range}} over a collection that should never be empty,
    //   adding a dummy {{else}} can help.
    ErrBranchEnd
```

```
// ErrEndContext: "... ends in a non-text context: ..."
// Examples:
//   <div
//   <div title="no close quote>
//   <script>f()
// Discussion:
//   Executed templates should produce a DocumentFragment of HTML.
//   Templates that end without closing tags will trigger this error.
//   Templates that should not be used in an HTML context or that
//   produce incomplete Fragments should not be executed directly.
//
//   {{define "main"}} <script>{{template "helper"}}</script> {{end}}
//   {{define "helper"}} document.write(' <div title=" ') {{end}}
//
//   "helper" does not produce a valid document fragment, so should
//   not be Executed directly.
ErrEndContext

// ErrNoSuchTemplate: "no such template ..."
// Examples:
//   {{define "main"}}<div {{template "attrs"}}>{{end}}
//   {{define "attrs"}}href="{{.URL}}"{{end}}
// Discussion:
//   Package html/template looks through template calls to compute the
//   context.
//   Here the {{.URL}} in "attrs" must be treated as a URL when called
//   from "main", but you will get this error if "attrs" is not defined
//   when "main" is parsed.
ErrNoSuchTemplate

// ErrOutputContext: "cannot compute output context for template ..."
// Examples:
//   {{define "t"}}{{if .T}}{{template "t" .T}}{{end}}{{.H}}",{{end}}
// Discussion:
//   A recursive template does not end in the same context in which it
//   starts, and a reliable output context cannot be computed.
//   Look for typos in the named template.
//   If the template should not be called in the named start context,
//   look for calls to that template in unexpected contexts.
//   Maybe refactor recursive templates to not be recursive.
ErrOutputContext

// ErrPartialCharset: "unfinished JS regexp charset in ..."
// Example:
//     <script>var pattern = /foo[{{.Chars}}]/</script>
// Discussion:
//   Package html/template does not support interpolation into regular
//   expression literal character sets.
ErrPartialCharset

// ErrPartialEscape: "unfinished escape sequence in ..."
// Example:
//   <script>alert("\{{.X}}")</script>
// Discussion:
	//   Package html/template does not support actions following a
	//   backslash.
	//   This is usually an error and there are better solutions; for
	//   example
	//     <script>alert("{{.X}}")</script>
	//   should work, and if {{.X}} is a partial escape sequence such as
	//   "xA0", mark the whole sequence as safe content: JSStr(`\xA0`)
	ErrPartialEscape

	// ErrRangeLoopReentry: "on range loop re-entry: ..."
	// Example:
	//   <script>var x = [{{range .}}'{{.}},{{end}}]</script>
	// Discussion:
	//   If an iteration through a range would cause it to end in a
	//   different context than an earlier pass, there is no single context.
	//   In the example, there is missing a quote, so it is not clear
	//   whether {{.}} is meant to be inside a JS string or in a JS value
	//   context. The second iteration would produce something like
	//
	//     <script>var x = ['firstValue,'secondValue]</script>
	ErrRangeLoopReentry

	// ErrSlashAmbig: '/' could start a division or regexp.
	// Example:
	//   <script>
	//     {{if .C}}var x = 1{{end}}
	//     /-{{.N}}/i.test(x) ? doThis : doThat();
	//   </script>
	// Discussion:
	//   The example above could produce `var x = 1/-2/i.test(s)...`
	//   in which the first '/' is a mathematical division operator or it
	//   could produce `/-2/i.test(s)` in which the first '/' starts a
	//   regexp literal.
	//   Look for missing semicolons inside branches, and maybe add
	//   parentheses to make it clear which interpretation you intend.
	ErrSlashAmbig

	// ErrPredefinedEscaper: "predefined escaper ... disallowed in template"
	// Example:
	//   <div class={{. | html}}>Hello<div>
	// Discussion:
	//   Package html/template already contextually escapes all pipelines to
	//   produce HTML output safe against code injection. Manually escaping
	//   pipeline output using the predefined escapers "html" or "urlquery" is
	//   unnecessary, and may affect the correctness or safety of the escaped
	//   pipeline output in Go 1.8 and earlier.
	//
	//   In most cases, such as the given example, this error can be resolved by
	//   simply removing the predefined escaper from the pipeline and letting the
	//   contextual autoescaper handle the escaping of the pipeline. In other
	//   instances, where the predefined escaper occurs in the middle of a
	//   pipeline where subsequent commands expect escaped input, e.g.
	//     {{.X | html | makeALink}}
	//   where makeALink does
```

```
    //     return `<a href="`+input+`">link</a>`
    //   consider refactoring the surrounding template to make use of the
    //   contextual autoescaper, i.e.
    //     <a href="{{.X}}">link</a>
    //
    //   To ease migration to Go 1.9 and beyond, "html" and "urlquery" will
    //   continue to be allowed as the last command in a pipeline. However, if the
    //   pipeline occurs in an unquoted attribute value context, "html" is
    //   disallowed. Avoid using "html" and "urlquery" entirely in new templates.
    ErrPredefinedEscaper
)
```

We define codes for each error that manifests while escaping templates, but escaped templates may also fail at runtime.

Output: "ZgotmplZ" Example:

```
<img src="{{.X}}">
where {{.X}} evaluates to `javascript:...`
```

Discussion:

```
"ZgotmplZ" is a special value that indicates that unsafe content reached a
CSS or URL context at runtime. The output of the example will be
  <img src="#ZgotmplZ">
If the data comes from a trusted source, use content types to exempt it
from filtering: URL(`javascript:...`).
```

type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) argument evaluates to non-nil during execution, execution terminates and Execute returns that error. FuncMap has the same base type as FuncMap in "text/template", copied here so clients need not import "text/template".

type HTML

```
type HTML string
```

HTML encapsulates a known safe HTML document fragment. It should not be used for HTML from a third-party, or HTML with unclosed tags or comments. The outputs of a sound HTML sanitizer and a template escaped by this package are fine for use with HTML.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type HTMLAttr

```
type HTMLAttr string
```

HTMLAttr encapsulates an HTML attribute from a trusted source, for example, ` dir="ltr"`.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type JS

```
type JS string
```

JS encapsulates a known safe EcmaScript5 Expression, for example, `(x + y * z())`. Template authors are responsible for ensuring that typed expressions do not break the intended precedence and that there is no statement/expression ambiguity as when passing an expression like "{ foo: bar() }\n['foo']()", which is both a valid Expression and a valid Program with a very different meaning.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

Using JS to include valid but untrusted JSON is not safe. A safe alternative is to parse the JSON with json.Unmarshal and then pass the resultant object into the template, where it will be converted to sanitized JSON when presented in a JavaScript context.

type JSStr

```
type JSStr string
```

JSStr encapsulates a sequence of characters meant to be embedded between quotes in a JavaScript expression. The string must match a series of StringCharacters:

```
StringCharacter :: SourceCharacter but not `\` or LineTerminator
                 | EscapeSequence
```

Note that LineContinuations are not allowed. JSStr("foo\\nbar") is fine, but JSStr("foo\\\nbar") is not.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type Srcset

```
type Srcset string
```

Srcset encapsulates a known safe srcset attribute (see https://w3c.github.io/html/semantics-embedded-content.html#element-attrdef-img-srcset).

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type Template

```
type Template struct {

    // The underlying template's parse tree, updated to be HTML-safe.
    Tree *parse.Tree
    // contains filtered or unexported fields
}
```

Template is a specialized Template from "text/template" that produces a safe HTML document fragment.

func Must

```
func Must(t *Template, err error) *Template
```

Must is a helper that wraps a call to a function returning (*Template, error) and panics if the error is non-nil. It is intended for use in variable initializations such as

```
var t = template.Must(template.New("name").Parse("html"))
```

func New

```
func New(name string) *Template
```

New allocates a new HTML template with the given name.

func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the (base) name and (parsed) contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results. For instance, ParseFiles("a/foo", "b/foo") stores "b/foo" as the template named "foo", while "a/foo" is unavailable.

func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

func (*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
```

AddParseTree creates a new template with the name and parse tree and associates it with t.

It returns an error if t or any associated template has already been executed.

func (*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

It returns an error if t has already been executed.

func (*Template) DefinedTemplates

```
func (t *Template) DefinedTemplates() string
```

DefinedTemplates returns a string listing the defined templates, prefixed by the string "; defined templates are: ". If there are none, it returns the empty string. Used to generate an error message.

func (*Template) Delims

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

Execute applies a parsed template to the specified data object, writing the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

func (*Template) Funcs

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs adds the elements of the argument map to the template's function map. It must be called before the template is parsed. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

func (*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

Lookup returns the template with the given name that is associated with t, or nil if there is no such template.

func (*Template) Name

```
func (t *Template) Name() string
```

Name returns the name of the template.

func (*Template) New

```
func (t *Template) New(name string) *Template
```

New allocates a new HTML template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a {{template}} action.

If a template with the given name already exists, the new HTML template will replace it. The existing template will be reset and disassociated with t.

func (*Template) Option

```
func (t *Template) Option(opt ...string) *Template
```

Option sets options for the template. Options are described by strings, either a simple string or "key=value". There can be at most one equals sign in an option string. If the option string is unrecognized or otherwise invalid, Option panics.

Known options:

missingkey: Control the behavior during execution if a map is indexed with a key that is not present in the map.

```
"missingkey=default" or "missingkey=invalid"
    The default behavior: Do nothing and continue execution.
    If printed, the result of the index operation is the string
    "<no value>".
"missingkey=zero"
    The operation returns the zero value for the map type's element.
"missingkey=error"
    Execution stops immediately with an error.
```

func (*Template) Parse

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse parses text as a template body for t. Named template definitions ({{define ...}} or {{block ...}} statements) in text define additional templates associated with t and are removed from the definition of t itself.

Templates can be redefined in successive calls to Parse, before the first use of Execute on t or any associated template. A template definition with a body containing only white space and comments is considered empty and will not replace an existing template's body. This allows using Parse to add new named template definitions without overwriting the main template body.

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

ParseFiles returns an error if t or any associated template has already been executed.

func (*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseGlob parses the template definitions in the files identified by the pattern and associates the resulting templates with t. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. ParseGlob is equivalent to calling t.ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

ParseGlob returns an error if t or any associated template has already been executed.

func (*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates returns a slice of the templates associated with t, including t itself.

type URL

```
type URL string
```

URL encapsulates a known safe URL or URL substring (see RFC 3986). A URL like `javascript:checkThatFormNotEditedBeforeLeavingPage()` from a trusted source should go in the page, but by default dynamic `javascript:` URLs are filtered out since they are a frequently exploited injection vector.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.