NAME

HTML::Form – Class that represents an HTML form element

VERSION

version 6.07

SYNOPSIS

```
use HTML::Form;
$form = HTML::Form->parse($html, $base_uri);
$form->value(query => "Perl");

use LWP::UserAgent;
$ua = LWP::UserAgent->new;
$response = $ua->request($form->click);
```

DESCRIPTION

Objects of the `HTML::Form` class represents a single HTML `<form> ... </form>` instance. A form consists of a sequence of inputs that usually have names, and which can take on various values. The state of a form can be tweaked and it can then be asked to provide HTTP::Request objects that can be passed to the **request()** method of LWP::UserAgent.

The following methods are available:

`@forms` = HTML::Form->parse(`$html_document`, `$base_uri`)
`@forms` = HTML::Form->parse(`$html_document`, base => `$base_uri`, `%opt`)
`@forms` = HTML::Form->parse(`$response`, `%opt`)

The **parse()** class method will parse an HTML document and build up `HTML::Form` objects for each <form> element found. If called in scalar context only returns the first <form>. Returns an empty list if there are no forms to be found.

The required arguments is the HTML document to parse ($html_document) and the URI used to retrieve the document ($base_uri). The base URI is needed to resolve relative action URIs. The provided HTML document should be a Unicode string (or US-ASCII).

By default HTML::Form assumes that the original document was UTF-8 encoded and thus encode forms that don't specify an explicit *accept-charset* as UTF-8. The charset assumed can be overridden by providing the `charset` option to **parse()**. It's a good idea to be explicit about this parameter as well, thus the recommended simplest invocation becomes:

```
my @forms = HTML::Form->parse(
    Encode::decode($encoding, $html_document_bytes),
    base => $base_uri,
    charset => $encoding,
);
```

If the document was retrieved with LWP then the response object provide methods to obtain a proper value for `base` and `charset`:

```
my $ua = LWP::UserAgent->new;
my $response = $ua->get("http://www.example.com/form.html");
my @forms = HTML::Form->parse($response->decoded_content,
    base => $response->base,
    charset => $response->content_charset,
);
```

In fact, the **parse()** method can parse from an HTTP::Response object directly, so the example above can be more conveniently written as:

```
my $ua = LWP::UserAgent->new;
my $response = $ua->get("http://www.example.com/form.html");
my @forms = HTML::Form->parse($response);
```

Note that any object that implements a **decoded_content()**, **base()** and **content_charset()** method with similar behaviour as HTTP::Response will do.

Additional options might be passed in to control how the parse method behaves. The following are all the options currently recognized:

`base => $uri`
: This is the URI used to retrieve the original document. This option is not optional ;–)

`charset => $str`
: Specify what charset the original document was encoded in. This is used as the default for accept_charset. If not provided this defaults to ‘‘UTF–8’’.

`verbose => $bool`
: Warn (print messages to STDERR) about any bad HTML form constructs found. You can trap these with `$SIG{__WARN__}`. The default is not to issue warnings.

`strict => $bool`
: Initialize any form objects with the given strict attribute. If the strict is turned on the methods that change values of the form will croak if you try to set illegal values or modify readonly fields. The default is not to be strict.

`$form`–>push_input(`$type`, \%attr, `$verbose`)
: This method adds additional inputs to the form. The first argument is the type of input (e.g. hidden, option, etc.). The second argument is a reference to a hash of the input attributes. The third argument is optional, and will issue warnings about unknown input types.

Example:

```
push_input( 'hidden', {
    name  => 'NewFormElement',
    id    => 'NewFormElementId',
    value => 'some value',
});
```

`$method` = `$form`–>method
`$form`–>method(`$new_method`)
: This method is gets/sets the *method* name used for the HTTP::Request generated. It is a string like ‘‘GET’’ or ‘‘POST’’.

`$action` = `$form`–>action
`$form`–>action(`$new_action`)
: This method gets/sets the URI which we want to apply the request *method* to.

`$enctype` = `$form`–>enctype
`$form`–>enctype(`$new_enctype`)
: This method gets/sets the encoding type for the form data. It is a string like ‘‘application/x–www–form–urlencoded’’ or ‘‘multipart/form–data’’.

`$accept` = `$form`–>accept_charset
`$form`–>accept_charset(`$new_accept`)
: This method gets/sets the list of charset encodings that the server processing the form accepts. Current implementation supports only one-element lists. Default value is ‘‘UNKNOWN’’ which we interpret as a request to use document charset as specified by the ’charset’ parameter of the **parse()** method.

`$value` = `$form`–>attr(`$name`)

`$form->attr( $name, $new_value )`
: This method give access to the original HTML attributes of the <form> tag. The `$name` should always be passed in lower case.

  Example:

  ```
  @f = HTML::Form->parse( $html, $foo );
  @f = grep $_->attr("id") eq "foo", @f;
  die "No form named 'foo' found" unless @f;
  $foo = shift @f;
  ```

`$bool = $form->strict`
`$form->strict( $bool )`
: Gets/sets the strict attribute of a form. If the strict is turned on the methods that change values of the form will croak if you try to set illegal values or modify readonly fields. The default is not to be strict.

`@inputs = $form->inputs`
: This method returns the list of inputs in the form. If called in scalar context it returns the number of inputs contained in the form. See "INPUTS" for what methods are available for the input objects returned.

`$input = $form->find_input( $selector )`
`$input = $form->find_input( $selector, $type )`
`$input = $form->find_input( $selector, $type, $index )`
`@inputs = $form->find_input( $selector )`
`@inputs = $form->find_input( $selector, $type )`
: This method is used to locate specific inputs within the form. All inputs that match the arguments given are returned. In scalar context only the first is returned, or `undef` if none match.

  If `$selector` is not `undef`, then the input's name, id, class attribute must match. A selector prefixed with '#' must match the id attribute of the input. A selector prefixed with '.' matches the class attribute. A selector prefixed with '^' or with no prefix matches the name attribute.

  If `$type` is not `undef`, then the input must have the specified type. The following type names are used: "text", "password", "hidden", "textarea", "file", "image", "submit", "radio", "checkbox" and "option".

  The `$index` is the sequence number of the input matched where 1 is the first. If combined with `$name` and/or `$type`, then it selects the *n*th input with the given name and/or type.

`$value = $form->value( $selector )`
`$form->value( $selector, $new_value )`
: The **value()** method can be used to get/set the value of some input. If strict is enabled and no input has the indicated name, then this method will croak.

  If multiple inputs have the same name, only the first one will be affected.

  The call:

  ```
  $form->value('foo')
  ```

  is basically a short-hand for:

  ```
  $form->find_input('foo')->value;
  ```

`@names = $form->param`
`@values = $form->param( $name )`
`$form->param( $name, $value, ... )`
`$form->param( $name, \@values )`
: Alternative interface to examining and setting the values of the form.

  If called without arguments then it returns the names of all the inputs in the form. The names will not repeat even if multiple inputs have the same name. In scalar context the number of different names is

returned.

If called with a single argument then it returns the value or values of inputs with the given name. If called in scalar context only the first value is returned. If no input exists with the given name, then `undef` is returned.

If called with 2 or more arguments then it will set values of the named inputs. This form will croak if no inputs have the given name or if any of the values provided does not fit. Values can also be provided as a reference to an array. This form will allow unsetting all values with the given name as well.

This interface resembles that of the **param()** function of the CGI module.

`$form->try_others( \&callback )`
: This method will iterate over all permutations of unvisited enumerated values (<select>, <radio>, <checkbox>) and invoke the callback for each. The callback is passed the `$form` as argument. The return value from the callback is ignored and the **try_others()** method itself does not return anything.

`$request = $form->make_request`
: Will return an HTTP::Request object that reflects the current setting of the form. You might want to use the **click()** method instead.

`$request = $form->click`
`$request = $form->click( $selector )`
`$request = $form->click( $x, $y )`
`$request = $form->click( $selector, $x, $y )`
: Will "click" on the first clickable input (which will be of type `submit` or `image`). The result of clicking is an HTTP::Request object that can then be passed to LWP::UserAgent if you want to obtain the server response.

If a `$selector` is specified, we will click on the first clickable input matching the selector, and the method will croak if no matching clickable input is found. If `$selector` is *not* specified, then it is ok if the form contains no clickable inputs. In this case the **click()** method returns the same request as the **make_request()** method would do. See description of the **find_input()** method above for how the `$selector` is specified.

If there are multiple clickable inputs with the same name, then there is no way to get the **click()** method of the `HTML::Form` to click on any but the first. If you need this you would have to locate the input with **find_input()** and invoke the **click()** method on the given input yourself.

A click coordinate pair can also be provided, but this only makes a difference if you clicked on an image. The default coordinate is (1,1). The upper-left corner of the image is (0,0), but some badly coded CGI scripts are known to not recognize this. Therefore (1,1) was selected as a safer default.

`@kw = $form->form`
: Returns the current setting as a sequence of key/value pairs. Note that keys might be repeated, which means that some values might be lost if the return values are assigned to a hash.

In scalar context this method returns the number of key/value pairs generated.

`$form->dump`
: Returns a textual representation of current state of the form. Mainly useful for debugging. If called in void context, then the dump is printed on STDERR.

INPUTS

An `HTML::Form` objects contains a sequence of *inputs*. References to the inputs can be obtained with the `$form->inputs` or `$form->find_input` methods.

Note that there is *not* a one-to-one correspondence between input *objects* and <input> *elements* in the HTML document. An input object basically represents a name/value pair, so when multiple HTML elements contribute to the same name/value pair in the submitted form they are combined.

The input elements that are mapped one-to-one are "text", "textarea", "password", "hidden", "file",

“image”, “submit” and “checkbox”. For the “radio” and “option” inputs the story is not as simple: All <input type=“radio”> elements with the same name will contribute to the same input radio object. The number of radio input objects will be the same as the number of distinct names used for the <input type=“radio”> elements. For a <select> element without the `multiple` attribute there will be one input object of type of “option”. For a <select multiple> element there will be one input object for each contained <option> element. Each one of these option objects will have the same name.

The following methods are available for the *input* objects:

`$input->`type
: Returns the type of this input. The type is one of the following strings: “text”, “password”, “hidden”, “textarea”, “file”, “image”, “submit”, “radio”, “checkbox” or “option”.

`$name` = `$input->`name
`$input->`name(`$new_name`)
: This method can be used to get/set the current name of the input.

`$input->`id
`$input->`class
: These methods can be used to get/set the current id or class attribute for the input.

`$input->`selected(`$selector`)
: Returns TRUE if the given selector matched the input. See the description of the **find_input()** method above for a description of the selector syntax.

`$value` = `$input->`value
`$input->`value(`$new_value`)
: This method can be used to get/set the current value of an input.

  If strict is enabled and the input only can take an enumerated list of values, then it is an error to try to set it to something else and the method will croak if you try.

  You will also be able to set the value of read-only inputs, but a warning will be generated if running under `perl -w`.

`$autocomplete` = `$input->`autocomplete
`$input->`autocomplete(`$new_autocomplete`)
: This method can be used to get/set the current value (if any) of `autcomplete` for the input.

`$input->`possible_values
: Returns a list of all values that an input can take. For inputs that do not have discrete values, this returns an empty list.

`$input->`other_possible_values
: Returns a list of all values not tried yet.

`$input->`value_names
: For some inputs the values can have names that are different from the values themselves. The number of names returned by this method will match the number of values reported by `$input->`possible_values.

  When setting values using the **value()** method it is also possible to use the value names in place of the value itself.

`$bool` = `$input->`readonly
`$input->`readonly(`$bool`)
: This method is used to get/set the value of the readonly attribute. You are allowed to modify the value of readonly inputs, but setting the value will generate some noise when warnings are enabled. Hidden fields always start out readonly.

`$bool` = `$input->`disabled

`$input->disabled( $bool )`
: This method is used to get/set the value of the disabled attribute. Disabled inputs do not contribute any key/value pairs for the form value.

`$input->form_name_value`
: Returns a (possible empty) list of key/value pairs that should be incorporated in the form value from this input.

`$input->check`
: Some input types represent toggles that can be turned on/off. This includes "checkbox" and "option" inputs. Calling this method turns this input on without having to know the value name. If the input is already on, then nothing happens.

  This has the same effect as:

```
$input->value($input->possible_values[1]);
```

  The input can be turned off with:

```
$input->value(undef);
```

`$input->click($form, $x, $y)`
: Some input types (currently "submit" buttons and "images") can be clicked to submit the form. The **click()** method returns the corresponding HTTP::Request object.

If the input is of type `file`, then it has these additional methods:

`$input->file`
: This is just an alias for the **value()** method. It sets the filename to read data from.

  For security reasons this field will never be initialized from the parsing of a form. This prevents the server from triggering stealth uploads of arbitrary files from the client machine.

`$filename = $input->filename`
`$input->filename( $new_filename )`
: This get/sets the filename reported to the server during file upload. This attribute defaults to the value reported by the **file()** method.

`$content = $input->content`
`$input->content( $new_content )`
: This get/sets the file content provided to the server during file upload. This method can be used if you do not want the content to be read from an actual file.

`@headers = $input->headers`
`input->headers($key => $value, .... )`
: This get/set additional header fields describing the file uploaded. This can for instance be used to set the `Content-Type` reported for the file.

SEE ALSO

LWP, LWP::UserAgent, HTML::Parser

AUTHOR

Gisle Aas <gisle@activestate.com>

COPYRIGHT AND LICENSE