NAME

URI – Uniform Resource Identifiers (absolute and relative)

SYNOPSIS

```
use URI;

$u1 = URI->new("http://www.perl.com");
$u2 = URI->new("foo", "http");
$u3 = $u2->abs($u1);
$u4 = $u3->clone;
$u5 = URI->new("HTTP://WWW.perl.com:80")->canonical;

$str = $u->as_string;
$str = "$u";

$scheme = $u->scheme;
$opaque = $u->opaque;
$path   = $u->path;
$frag   = $u->fragment;

$u->scheme("ftp");
$u->host("ftp.perl.com");
$u->path("cpan/");
```

DESCRIPTION

This module implements the URI class. Objects of this class represent ‘‘Uniform Resource Identifier references’’ as specified in RFC 2396 (and updated by RFC 2732).

A Uniform Resource Identifier is a compact string of characters that identifies an abstract or physical resource. A Uniform Resource Identifier can be further classified as either a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). The distinction between URL and URN does not matter to the URI class interface. A ‘‘URI-reference’’ is a URI that may have additional information attached in the form of a fragment identifier.

An absolute URI reference consists of three parts: a *scheme*, a *scheme-specific part* and a *fragment* identifier. A subset of URI references share a common syntax for hierarchical namespaces. For these, the scheme-specific part is further broken down into *authority*, *path* and *query* components. These URIs can also take the form of relative URI references, where the scheme (and usually also the authority) component is missing, but implied by the context of the URI reference. The three forms of URI reference syntax are summarized as follows:

```
<scheme>:<scheme-specific-part>#<fragment>
<scheme>://<authority><path>?<query>#<fragment>
<path>?<query>#<fragment>
```

The components into which a URI reference can be divided depend on the *scheme*. The URI class provides methods to get and set the individual components. The methods available for a specific URI object depend on the scheme.

CONSTRUCTORS

The following methods construct new URI objects:

$uri = URI–>new($str)
$uri = URI–>new($str, $scheme)

Constructs a new URI object. The string representation of a URI is given as argument, together with an optional scheme specification. Common URI wrappers like "" and <>, as well as leading and trailing white space, are automatically removed from the $str argument before it is processed further.

The constructor determines the scheme, maps this to an appropriate URI subclass, constructs a new object of that class and returns it.

If the scheme isn't one of those that URI recognizes, you still get an URI object back that you can access the generic methods on. The `$uri->has_recognized_scheme` method can be used to test for this.

The `$scheme` argument is only used when `$str` is a relative URI. It can be either a simple string that denotes the scheme, a string containing an absolute URI reference, or an absolute `URI` object. If no `$scheme` is specified for a relative URI `$str`, then `$str` is simply treated as a generic URI (no scheme-specific methods available).

The set of characters available for building URI references is restricted (see URI::Escape). Characters outside this set are automatically escaped by the URI constructor.

`$uri` = URI->new_abs(`$str`, `$base_uri`)
Constructs a new absolute URI object. The `$str` argument can denote a relative or absolute URI. If relative, then it is absolutized using `$base_uri` as base. The `$base_uri` must be an absolute URI.

`$uri` = URI::file->new(`$filename`)
`$uri` = URI::file->new(`$filename`, `$os`)
Constructs a new *file* URI from a file name. See URI::file.

`$uri` = URI::file->new_abs(`$filename`)
`$uri` = URI::file->new_abs(`$filename`, `$os`)
Constructs a new absolute *file* URI from a file name. See URI::file.

`$uri` = URI::file->cwd
Returns the current working directory as a *file* URI. See URI::file.

`$uri->clone`
Returns a copy of the `$uri`.

COMMON METHODS

The methods described in this section are available for all `URI` objects.

Methods that give access to components of a URI always return the old value of the component. The value returned is `undef` if the component was not present. There is generally a difference between a component that is empty (represented as `""`) and a component that is missing (represented as `undef`). If an accessor method is given an argument, it updates the corresponding component in addition to returning the old value of the component. Passing an undefined argument removes the component (if possible). The description of each accessor method indicates whether the component is passed as an escaped (percent-encoded) or an unescaped string. A component that can be further divided into sub-parts are usually passed escaped, as unescaping might change its semantics.

The common methods available for all URI are:

`$uri->scheme`
`$uri->scheme( $new_scheme )`
Sets and returns the scheme part of the `$uri`. If the `$uri` is relative, then `$uri->scheme` returns `undef`. If called with an argument, it updates the scheme of `$uri`, possibly changing the class of `$uri`, and returns the old scheme value. The method croaks if the new scheme name is illegal; a scheme name must begin with a letter and must consist of only US-ASCII letters, numbers, and a few special marks: ".", "+", "-". This restriction effectively means that the scheme must be passed unescaped. Passing an undefined argument to the scheme method makes the URI relative (if possible).

Letter case does not matter for scheme names. The string returned by `$uri->scheme` is always lowercase. If you want the scheme just as it was written in the URI in its original case, you can use the `$uri->_scheme` method instead.

`$uri->has_recognized_scheme`
Returns TRUE if the URI scheme is one that URI recognizes.

It will also be TRUE for relative URLs where a recognized scheme was provided to the constructor, even if `$uri->scheme` returns `undef` for these.

`$uri->opaque`
`$uri->opaque( $new_opaque )`
: Sets and returns the scheme-specific part of the `$uri` (everything between the scheme and the fragment) as an escaped string.

`$uri->path`
`$uri->path( $new_path )`
: Sets and returns the same value as `$uri->opaque` unless the URI supports the generic syntax for hierarchical namespaces. In that case the generic method is overridden to set and return the part of the URI between the *host name* and the *fragment*.

`$uri->fragment`
`$uri->fragment( $new_frag )`
: Returns the fragment identifier of a URI reference as an escaped string.

`$uri->as_string`
: Returns a URI object to a plain ASCII string. URI objects are also converted to plain strings automatically by overloading. This means that `$uri` objects can be used as plain strings in most Perl constructs.

`$uri->as_iri`
: Returns a Unicode string representing the URI. Escaped UTF-8 sequences representing non-ASCII characters are turned into their corresponding Unicode code point.

`$uri->canonical`
: Returns a normalized version of the URI. The rules for normalization are scheme-dependent. They usually involve lowercasing the scheme and Internet host name components, removing the explicit port specification if it matches the default port, uppercasing all escape sequences, and unescaping octets that can be better represented as plain characters.

  For efficiency reasons, if the `$uri` is already in normalized form, then a reference to it is returned instead of a copy.

`$uri->eq( $other_uri )`
`URI::eq( $first_uri, $other_uri )`
: Tests whether two URI references are equal. URI references that normalize to the same string are considered equal. The method can also be used as a plain function which can also test two string arguments.

  If you need to test whether two `URI` object references denote the same object, use the '==' operator.

`$uri->abs( $base_uri )`
: Returns an absolute URI reference. If `$uri` is already absolute, then a reference to it is simply returned. If the `$uri` is relative, then a new absolute URI is constructed by combining the `$uri` and the `$base_uri`, and returned.

`$uri->rel( $base_uri )`
: Returns a relative URI reference if it is possible to make one that denotes the same resource relative to `$base_uri`. If not, then `$uri` is simply returned.

`$uri->secure`
: Returns a TRUE value if the URI is considered to point to a resource on a secure channel, such as an SSL or TLS encrypted one.

GENERIC METHODS

The following methods are available to schemes that use the common/generic syntax for hierarchical namespaces. The descriptions of schemes below indicate which these are. Unrecognized schemes are assumed to support the generic syntax, and therefore the following methods:

`$uri->authority`

`$uri->authority( $new_authority )`
: Sets and returns the escaped authority component of the `$uri`.

`$uri->path`
`$uri->path( $new_path )`
: Sets and returns the escaped path component of the `$uri` (the part between the host name and the query or fragment). The path can never be undefined, but it can be the empty string.

`$uri->path_query`
`$uri->path_query( $new_path_query )`
: Sets and returns the escaped path and query components as a single entity. The path and the query are separated by a "?" character, but the query can itself contain "?".

`$uri->path_segments`
`$uri->path_segments( $segment, ... )`
: Sets and returns the path. In a scalar context, it returns the same value as `$uri->path`. In a list context, it returns the unescaped path segments that make up the path. Path segments that have parameters are returned as an anonymous array. The first element is the unescaped path segment proper; subsequent elements are escaped parameter strings. Such an anonymous array uses overloading so it can be treated as a string too, but this string does not include the parameters.

  Note that absolute paths have the empty string as their first *path_segment*, i.e. the *path* `/foo/bar` have 3 *path_segments*; "", "foo" and "bar".

`$uri->query`
`$uri->query( $new_query )`
: Sets and returns the escaped query component of the `$uri`.

`$uri->query_form`
`$uri->query_form( $key1 => $val1, $key2 => $val2, ... )`
`$uri->query_form( $key1 => $val1, $key2 => $val2, ..., $delim )`
`$uri->query_form( \@key_value_pairs )`
`$uri->query_form( \@key_value_pairs, $delim )`
`$uri->query_form( \%hash )`
`$uri->query_form( \%hash, $delim )`
: Sets and returns query components that use the *application/x-www-form-urlencoded* format. Key/value pairs are separated by "&", and the key is separated from the value by a "=" character.

  The form can be set either by passing separate key/value pairs, or via an array or hash reference. Passing an empty array or an empty hash removes the query component, whereas passing no arguments at all leaves the component unchanged. The order of keys is undefined if a hash reference is passed. The old value is always returned as a list of separate key/value pairs. Assigning this list to a hash is unwise as the keys returned might repeat.

  The values passed when setting the form can be plain strings or references to arrays of strings. Passing an array of values has the same effect as passing the key repeatedly with one value at a time. All the following statements have the same effect:

```
$uri->query_form(foo => 1, foo => 2);
$uri->query_form(foo => [1, 2]);
$uri->query_form([ foo => 1, foo => 2 ]);
$uri->query_form([ foo => [1, 2] ]);
$uri->query_form({ foo => [1, 2] });
```

  The `$delim` parameter can be passed as ";" to force the key/value pairs to be delimited by ";" instead of "&" in the query string. This practice is often recommended for URLs embedded in HTML or XML documents as this avoids the trouble of escaping the "&" character. You might also set the `$URI::DEFAULT_QUERY_FORM_DELIMITER` variable to ";" for the same global effect.

  The `URI::QueryParam` module can be loaded to add further methods to manipulate the form of a

URI. See URI::QueryParam for details.

`$uri->query_keywords`
`$uri->query_keywords( $keywords, ... )`
`$uri->query_keywords( \@keywords )`
Sets and returns query components that use the keywords separated by "+" format.

The keywords can be set either by passing separate keywords directly or by passing a reference to an array of keywords. Passing an empty array removes the query component, whereas passing no arguments at all leaves the component unchanged. The old value is always returned as a list of separate words.

SERVER METHODS

For schemes where the *authority* component denotes an Internet host, the following methods are available in addition to the generic methods.

`$uri->userinfo`
`$uri->userinfo( $new_userinfo )`
Sets and returns the escaped userinfo part of the authority component.

For some schemes this is a user name and a password separated by a colon. This practice is not recommended. Embedding passwords in clear text (such as URI) has proven to be a security risk in almost every case where it has been used.

`$uri->host`
`$uri->host( $new_host )`
Sets and returns the unescaped hostname.

If the `$new_host` string ends with a colon and a number, then this number also sets the port.

For IPv6 addresses the brackets around the raw address is removed in the return value from `$uri->host`. When setting the host attribute to an IPv6 address you can use a raw address or one enclosed in brackets. The address needs to be enclosed in brackets if you want to pass in a new port value as well.

`$uri->ihost`
Returns the host in Unicode form. Any IDNA A-labels are turned into U-labels.

`$uri->port`
`$uri->port( $new_port )`
Sets and returns the port. The port is a simple integer that should be greater than 0.

If a port is not specified explicitly in the URI, then the URI scheme's default port is returned. If you don't want the default port substituted, then you can use the `$uri->_port` method instead.

`$uri->host_port`
`$uri->host_port( $new_host_port )`
Sets and returns the host and port as a single unit. The returned value includes a port, even if it matches the default port. The host part and the port part are separated by a colon: ":".

For IPv6 addresses the bracketing is preserved; thus URI->new("http://[::1]/")->host_port returns "[::1]:80". Contrast this with `$uri->host` which will remove the brackets.

`$uri->default_port`
Returns the default port of the URI scheme to which `$uri` belongs. For *http* this is the number 80, for *ftp* this is the number 21, etc. The default port for a scheme can not be changed.

SCHEME-SPECIFIC SUPPORT

Scheme-specific support is provided for the following URI schemes. For `URI` objects that do not belong to one of these, you can only use the common and generic methods.

data:
The *data* URI scheme is specified in RFC 2397. It allows inclusion of small data items as ''immediate'' data, as if it had been included externally.

`URI` objects belonging to the data scheme support the common methods and two new methods to access their scheme-specific components: `$uri->`media_type and `$uri->`data. See URI::data for details.

file:
An old specification of the *file* URI scheme is found in RFC 1738. A new RFC 2396 based specification in not available yet, but file URI references are in common use.

`URI` objects belonging to the file scheme support the common and generic methods. In addition, they provide two methods for mapping file URIs back to local file names; `$uri->`file and `$uri->`dir. See URI::file for details.

ftp: An old specification of the *ftp* URI scheme is found in RFC 1738. A new RFC 2396 based specification in not available yet, but ftp URI references are in common use.

`URI` objects belonging to the ftp scheme support the common, generic and server methods. In addition, they provide two methods for accessing the userinfo sub-components: `$uri->`user and `$uri->`password.

gopher:
The *gopher* URI scheme is specified in <draft-murali-url-gopher-1996-12-04> and will hopefully be available as a RFC 2396 based specification.

`URI` objects belonging to the gopher scheme support the common, generic and server methods. In addition, they support some methods for accessing gopher-specific path components: `$uri->`gopher_type, `$uri->`selector, `$uri->`search, `$uri->`string.

http:
The *http* URI scheme is specified in RFC 2616. The scheme is used to reference resources hosted by HTTP servers.

`URI` objects belonging to the http scheme support the common, generic and server methods.

https:
The *https* URI scheme is a Netscape invention which is commonly implemented. The scheme is used to reference HTTP servers through SSL connections. Its syntax is the same as http, but the default port is different.

ldap:
The *ldap* URI scheme is specified in RFC 2255. LDAP is the Lightweight Directory Access Protocol. An ldap URI describes an LDAP search operation to perform to retrieve information from an LDAP directory.

`URI` objects belonging to the ldap scheme support the common, generic and server methods as well as ldap-specific methods: `$uri->`dn, `$uri->`attributes, `$uri->`scope, `$uri->`filter, `$uri->`extensions. See URI::ldap for details.

ldapi:
Like the *ldap* URI scheme, but uses a UNIX domain socket. The server methods are not supported, and the local socket path is available as `$uri->`un_path. The *ldapi* scheme is used by the OpenLDAP package. There is no real specification for it, but it is mentioned in various OpenLDAP manual pages.

ldaps:
Like the *ldap* URI scheme, but uses an SSL connection. This scheme is deprecated, as the preferred way is to use the *start_tls* mechanism.

mailto:
The *mailto* URI scheme is specified in RFC 2368. The scheme was originally used to designate the Internet mailing address of an individual or service. It has (in RFC 2368) been extended to allow

setting of other mail header fields and the message body.

`URI` objects belonging to the mailto scheme support the common methods and the generic query methods. In addition, they support the following mailto-specific methods: `$uri->`to, `$uri->`headers.

Note that the "foo@example.com" part of a mailto is *not* the `userinfo` and `host` but instead the `path`. This allows a mailto URI to contain multiple comma separated email addresses.

mms:
The *mms* URL specification can be found at <http://sdp.ppona.com/>. `URI` objects belonging to the mms scheme support the common, generic, and server methods, with the exception of userinfo and query-related sub-components.

news:
The *news*, *nntp* and *snews* URI schemes are specified in <draft-gilman-news-url-01> and will hopefully be available as an RFC 2396 based specification soon.

`URI` objects belonging to the news scheme support the common, generic and server methods. In addition, they provide some methods to access the path: `$uri->`group and `$uri->`message.

nntp:
See *news* scheme.

pop:
The *pop* URI scheme is specified in RFC 2384. The scheme is used to reference a POP3 mailbox.

`URI` objects belonging to the pop scheme support the common, generic and server methods. In addition, they provide two methods to access the userinfo components: `$uri->`user and `$uri->`auth

rlogin:
An old specification of the *rlogin* URI scheme is found in RFC 1738. `URI` objects belonging to the rlogin scheme support the common, generic and server methods.

rtsp:
The *rtsp* URL specification can be found in section 3.2 of RFC 2326. `URI` objects belonging to the rtsp scheme support the common, generic, and server methods, with the exception of userinfo and query-related sub-components.

rtspu:
The *rtspu* URI scheme is used to talk to RTSP servers over UDP instead of TCP. The syntax is the same as rtsp.

rsync:
Information about rsync is available from <http://rsync.samba.org/>. `URI` objects belonging to the rsync scheme support the common, generic and server methods. In addition, they provide methods to access the userinfo sub-components: `$uri->`user and `$uri->`password.

sip:
The *sip* URI specification is described in sections 19.1 and 25 of RFC 3261. `URI` objects belonging to the sip scheme support the common, generic, and server methods with the exception of path related sub-components. In addition, they provide two methods to get and set *sip* parameters: `$uri->`params_form and `$uri->`params.

sips:
See *sip* scheme. Its syntax is the same as sip, but the default port is different.

snews:
See *news* scheme. Its syntax is the same as news, but the default port is different.

telnet:
An old specification of the *telnet* URI scheme is found in RFC 1738. `URI` objects belonging to the telnet scheme support the common, generic and server methods.

tn3270:
These URIs are used like *telnet* URIs but for connections to IBM mainframes. `URI` objects belonging to the tn3270 scheme support the common, generic and server methods.

ssh:
Information about ssh is available at <http://www.openssh.com/>. `URI` objects belonging to the ssh scheme support the common, generic and server methods. In addition, they provide methods to access the userinfo sub-components: `$uri->`user and `$uri->`password.

sftp:
`URI` objects belonging to the sftp scheme support the common, generic and server methods. In addition, they provide methods to access the userinfo sub-components: `$uri->`user and `$uri->`password.

urn:
The syntax of Uniform Resource Names is specified in RFC 2141. `URI` objects belonging to the urn scheme provide the common methods, and also the methods `$uri->`nid and `$uri->`nss, which return the Namespace Identifier and the Namespace-Specific String respectively.

The Namespace Identifier basically works like the Scheme identifier of URIs, and further divides the URN namespace. Namespace Identifier assignments are maintained at <http://www.iana.org/assignments/urn-namespaces>.

Letter case is not significant for the Namespace Identifier. It is always returned in lower case by the `$uri->`nid method. The `$uri->`_nid method can be used if you want it in its original case.

urn:**isbn**:
The `urn:isbn:` namespace contains International Standard Book Numbers (ISBNs) and is described in RFC 3187. A `URI` object belonging to this namespace has the following extra methods (if the Business::ISBN module is available): `$uri->`isbn, `$uri->`isbn_publisher_code, `$uri->`isbn_group_code (formerly isbn_country_code, which is still supported by issues a deprecation warning), `$uri->`isbn_as_ean.

urn:**oid**:
The `urn:oid:` namespace contains Object Identifiers (OIDs) and is described in RFC 3061. An object identifier consists of sequences of digits separated by dots. A `URI` object belonging to this namespace has an additional method called `$uri->`oid that can be used to get/set the oid value. In a list context, oid numbers are returned as separate elements.

CONFIGURATION VARIABLES

The following configuration variables influence how the class and its methods behave:

`$URI::ABS_ALLOW_RELATIVE_SCHEME`
Some older parsers used to allow the scheme name to be present in the relative URL if it was the same as the base URL scheme. RFC 2396 says that this should be avoided, but you can enable this old behaviour by setting the `$URI::ABS_ALLOW_RELATIVE_SCHEME` variable to a TRUE value. The difference is demonstrated by the following examples:

```
URI->new("http:foo")->abs("http://host/a/b")
    ==>  "http:foo"

local $URI::ABS_ALLOW_RELATIVE_SCHEME = 1;
URI->new("http:foo")->abs("http://host/a/b")
    ==>  "http:/host/a/foo"
```

`$URI::ABS_REMOTE_LEADING_DOTS`
You can also have the **abs()** method ignore excess "../" segments in the relative URI by setting `$URI::ABS_REMOTE_LEADING_DOTS` to a TRUE value. The difference is demonstrated by the following examples:

```
URI->new("../../../foo")->abs("http://host/a/b")
    ==> "http://host/../../foo"

local $URI::ABS_REMOTE_LEADING_DOTS = 1;
URI->new("../../../foo")->abs("http://host/a/b")
    ==> "http://host/foo"
```

`$URI::DEFAULT_QUERY_FORM_DELIMITER`

This value can be set to ";" to have the query form `key=value` pairs delimited by ";" instead of "&" which is the default.

BUGS

There are some things that are not quite right:

- Using regexp variables like `$1` directly as arguments to the URI accessor methods does not work too well with current perl implementations. I would argue that this is actually a bug in perl. The workaround is to quote them. Example:

  ```
  /(...)/ || die;
  $u->query("$1");
  ```

- The escaping (percent encoding) of chars in the 128 .. 255 range passed to the URI constructor or when setting URI parts using the accessor methods depend on the state of the internal UTF8 flag (see utf8::is_utf8) of the string passed. If the UTF8 flag is set the UTF-8 encoded version of the character is percent encoded. If the UTF8 flag isn't set the Latin-1 version (byte) of the character is percent encoded. This basically exposes the internal encoding of Perl strings.

PARSING URIs WITH REGEXP

As an alternative to this module, the following (official) regular expression can be used to decode a URI:

```
my($scheme, $authority, $path, $query, $fragment) =
$uri =~ m|(?:([^:/?#]+):)?(?://([^/?#]*))?([^?#]*)(?:\?([^#]*))?(?:#(.*))?|;
```

The `URI::Split` module provides the function **uri_split()** as a readable alternative.

SEE ALSO

URI::file, URI::WithBase, URI::QueryParam, URI::Escape, URI::Split, URI::Heuristic

RFC 2396: "Uniform Resource Identifiers (URI): Generic Syntax", Berners-Lee, Fielding, Masinter, August 1998.

<http://www.iana.org/assignments/uri-schemes>

<http://www.iana.org/assignments/urn-namespaces>

<http://www.w3.org/Addressing/>

COPYRIGHT

AUTHORS / ACKNOWLEDGMENTS

This module is based on the `URI::URL` module, which in turn was (distantly) based on the `wwwurl.pl` code in the libwww-perl for perl4 developed by Roy Fielding, as part of the Arcadia project at the University of California, Irvine, with contributions from Brooks Cutter.

`URI::URL` was developed by Gisle Aas, Tim Bunce, Roy Fielding and Martijn Koster with input from other people on the libwww-perl mailing list.

`URI` and related subclasses was developed by Gisle Aas.