URL

> *Stability: 2 - Stable*

The `url` module provides utilities for URL resolution and parsing. It can be accessed using:

```
import url from 'url';
```

```
const url = require('url');
```

URL strings and URL objects

A URL string is a structured string containing multiple meaningful components. When parsed, a URL object is returned containing properties for each of these components.

The `url` module provides two APIs for working with URLs: a legacy API that is Node.js specific, and a newer API that implements the same WHATWG URL Standard used by web browsers.

A comparison between the WHATWG and Legacy APIs is provided below. Above the URL `'https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'`, properties of an object returned by the legacy `url.parse()` are shown. Below it are properties of a WHATWG `URL` object.

WHATWG URL's `origin` property includes `protocol` and `host`, but not `username` or `password`.

```
┌──────────────────────────────────────────────────────────────────────────────────
│                                               href
│
├──────────┬──┬─────────────────────┬────────────────────────┬───────────────────────
│ protocol │  │        auth         │          host          │            path
│ hash    │
│          │  │                     │                        │
├─────────────────┬──────┼──────────┬────────────────┤       │
│          │  │                     │    hostname     │ port │ pathname │      search
│       │
│          │  │                     │                 │      │
├─┬─────────────────┤       │
│          │  │                     │                 │      │          │ │    query
│       │
"  https:   //    user   :   pass   @ sub.example.com : 8080   /p/a/t/h  ?
query=string   #hash "
│          │  │          │          │    hostname     │ port │          │
│       │
│          │  │          │          ├─────────────────┴──────┤          │
│       │
│ protocol │  │ username │ password │          host          │          │
│       │
├──────────┴──┼──────────┴──────────┼────────────────────────┤          │
│       │
│   origin    │                     │         origin          │ pathname │      search
│ hash  │
├─────────────┴─────────────────────┴─────────────────────────┴──────────┴────────────────
│                                           href
│
└─────────────────────────────────────────────────────────────────────────────────────────

(All spaces in the "" line should be ignored. They are purely for formatting.)
```

Parsing the URL string using the WHATWG API:

```
const myURL =
  new URL('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

Parsing the URL string using the Legacy API:

```
import url from 'url';
const myURL =
  url.parse('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

```
const url = require('url');
const myURL =
  url.parse('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

Constructing a URL from component parts and getting the constructed string

It is possible to construct a WHATWG URL from component parts using either the property setters or a template literal string:

```
const myURL = new URL('https://example.org');
myURL.pathname = '/a/b/c';
myURL.search = '?d=e';
myURL.hash = '#fgh';
```

```
const pathname = '/a/b/c';
const search = '?d=e';
const hash = '#fgh';
const myURL = new URL(`https://example.org${pathname}${search}${hash}`);
```

To get the constructed URL string, use the `href` property accessor:

```
console.log(myURL.href);
```

The WHATWG URL API

Class: `URL`

Browser-compatible `URL` class, implemented by following the WHATWG URL Standard. Examples of parsed URLs may be found in the Standard itself. The `URL` class is also available on the global object.

In accordance with browser conventions, all properties of `URL` objects are implemented as getters and setters on the class prototype, rather than as data properties on the object itself. Thus, unlike legacy `urlObject`s, using the `delete` keyword on any properties of `URL` objects (e.g. `delete myURL.protocol`, `delete myURL.pathname`, etc) has no effect but will still return `true`.

`new URL(input[, base])`

- `input` {string} The absolute or relative input URL to parse. If `input` is relative, then `base` is required. If `input` is absolute, the `base` is ignored. If `input` is not a string, it is converted to a string first.
- `base` {string} The base URL to resolve against if the `input` is not absolute. If `base` is not a string, it is converted to a string first.

Creates a new `URL` object by parsing the `input` relative to the `base`. If `base` is passed as a string, it will be parsed equivalent to `new URL(base)`.

```
const myURL = new URL('/foo', 'https://example.org/');
// https://example.org/foo
```

The URL constructor is accessible as a property on the global object. It can also be imported from the built-in url module:

```
import { URL } from 'url';
console.log(URL === globalThis.URL); // Prints 'true'.
```

```
console.log(URL === require('url').URL); // Prints 'true'.
```

A `TypeError` will be thrown if the `input` or `base` are not valid URLs. Note that an effort will be made to coerce the given values into strings. For instance:

```
const myURL = new URL({ toString: () => 'https://example.org/' });
// https://example.org/
```

Unicode characters appearing within the host name of `input` will be automatically converted to ASCII using the Punycode algorithm.

```
const myURL = new URL('https://測試');
// https://xn--g6w251d/
```

This feature is only available if the `node` executable was compiled with ICU enabled. If not, the domain names are passed through unchanged.

In cases where it is not known in advance if `input` is an absolute URL and a `base` is provided, it is advised to validate that the `origin` of the `URL` object is what is expected.

```
let myURL = new URL('http://Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https://Example.com/', 'https://example.org/');
// https://example.com/

myURL = new URL('foo://Example.com/', 'https://example.org/');
// foo://Example.com/

myURL = new URL('http:Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https:Example.com/', 'https://example.org/');
// https://example.org/Example.com/

myURL = new URL('foo:Example.com/', 'https://example.org/');
// foo:Example.com/
```

`url.hash`

- {string}

Gets and sets the fragment portion of the URL.

```
const myURL = new URL('https://example.org/foo#bar');
console.log(myURL.hash);
// Prints #bar

myURL.hash = 'baz';
console.log(myURL.href);
// Prints https://example.org/foo#baz
```

Invalid URL characters included in the value assigned to the `hash` property are percent-encoded. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.host`

- {string}

Gets and sets the host portion of the URL.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.host);
// Prints example.org:81

myURL.host = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:82/foo
```

Invalid host values assigned to the `host` property are ignored.

`url.hostname`

- {string}

Gets and sets the host name portion of the URL. The key difference between `url.host` and `url.hostname` is that `url.hostname` does *not* include the port.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.hostname);
// Prints example.org

// Setting the hostname does not change the port
myURL.hostname = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:81/foo

// Use myURL.host to change the hostname and port
myURL.host = 'example.org:82';
console.log(myURL.href);
// Prints https://example.org:82/foo
```

Invalid host name values assigned to the `hostname` property are ignored.

`url.href`

- {string}

Gets and sets the serialized URL.

```
const myURL = new URL('https://example.org/foo');
console.log(myURL.href);
// Prints https://example.org/foo

myURL.href = 'https://example.com/bar';
console.log(myURL.href);
// Prints https://example.com/bar
```

Getting the value of the `href` property is equivalent to calling `url.toString()`.

Setting the value of this property to a new value is equivalent to creating a new `URL` object using `new URL(value)`. Each of the `URL` object's properties will be modified.

If the value assigned to the `href` property is not a valid URL, a `TypeError` will be thrown.

`url.origin`

- {string}

Gets the read-only serialization of the URL's origin.

```
const myURL = new URL('https://example.org/foo/bar?baz');
console.log(myURL.origin);
// Prints https://example.org
```

```
const idnURL = new URL('https://測試');
console.log(idnURL.origin);
// Prints https://xn--g6w251d

console.log(idnURL.hostname);
// Prints xn--g6w251d
```

`url.password`

- {string}

Gets and sets the password portion of the URL.

```
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.password);
// Prints xyz

myURL.password = '123';
console.log(myURL.href);
// Prints https://abc:123@example.com
```

Invalid URL characters included in the value assigned to the `password` property are percent-encoded. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.pathname`

- {string}

Gets and sets the path portion of the URL.

```
const myURL = new URL('https://example.org/abc/xyz?123');
console.log(myURL.pathname);
// Prints /abc/xyz

myURL.pathname = '/abcdef';
console.log(myURL.href);
// Prints https://example.org/abcdef?123
```

Invalid URL characters included in the value assigned to the `pathname` property are percent-encoded. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.port`

- {string}

Gets and sets the port portion of the URL.

The port value may be a number or a string containing a number in the range `0` to `65535` (inclusive). Setting the value to the default port of the `URL` objects given `protocol` will result in the `port` value becoming the empty string (`''`).

The port value can be an empty string in which case the port depends on the protocol/scheme:

protocol	port
"ftp"	21
"file"	
"http"	80
"https"	443
"ws"	80
"wss"	443

Upon assigning a value to the port, the value will first be converted to a string using `.toString()`.

If that string is invalid but it begins with a number, the leading number is assigned to `port`. If the number lies outside the range denoted above, it is ignored.

```
const myURL = new URL('https://example.org:8888');
console.log(myURL.port);
// Prints 8888

// Default ports are automatically transformed to the empty string
// (HTTPS protocol's default port is 443)
myURL.port = '443';
console.log(myURL.port);
// Prints the empty string
console.log(myURL.href);
// Prints https://example.org/

myURL.port = 1234;
console.log(myURL.port);
// Prints 1234
console.log(myURL.href);
// Prints https://example.org:1234/

// Completely invalid port strings are ignored
myURL.port = 'abcd';
console.log(myURL.port);
// Prints 1234

// Leading numbers are treated as a port number
myURL.port = '5678abcd';
console.log(myURL.port);
// Prints 5678

// Non-integers are truncated
myURL.port = 1234.5678;
console.log(myURL.port);
// Prints 1234
// Out-of-range numbers which are not represented in scientific notation
// will be ignored.
myURL.port = 1e10; // 10000000000, will be range-checked as described below
console.log(myURL.port);
// Prints 1234
```

Numbers which contain a decimal point, such as floating-point numbers or numbers in scientific notation, are not an exception to this rule. Leading numbers up to the decimal point will be set as the URL's port, assuming they are valid:

```
myURL.port = 4.567e21;
console.log(myURL.port);
// Prints 4 (because it is the leading number in the string '4.567e21')
```

`url.protocol`

- {string}

Gets and sets the protocol portion of the URL.

```
const myURL = new URL('https://example.org');
console.log(myURL.protocol);
// Prints https:

myURL.protocol = 'ftp';
console.log(myURL.href);
// Prints ftp://example.org/
```

Invalid URL protocol values assigned to the `protocol` property are ignored.

Special schemes

The WHATWG URL Standard considers a handful of URL protocol schemes to be *special* in terms of how they are parsed and serialized. When a URL is parsed using one of these special protocols, the `url.protocol` property may be changed to another special protocol but cannot be changed to a non-special protocol, and vice versa.

For instance, changing from `http` to `https` works:

```
const u = new URL('http://example.org');
u.protocol = 'https';
console.log(u.href);
// https://example.org
```

However, changing from `http` to a hypothetical `fish` protocol does not because the new protocol is not special.

```
const u = new URL('http://example.org');
u.protocol = 'fish';
console.log(u.href);
// http://example.org
```

Likewise, changing from a non-special protocol to a special protocol is also not permitted:

```
const u = new URL('fish://example.org');
u.protocol = 'http';
console.log(u.href);
// fish://example.org
```

According to the WHATWG URL Standard, special protocol schemes are `ftp`, `file`, `http`, `https`, `ws`, and `wss`.

`url.search`

- {string}

Gets and sets the serialized query portion of the URL.

```
const myURL = new URL('https://example.org/abc?123');
console.log(myURL.search);
// Prints ?123

myURL.search = 'abc=xyz';
console.log(myURL.href);
// Prints https://example.org/abc?abc=xyz
```

Any invalid URL characters appearing in the value assigned the `search` property will be percent-encoded. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.searchParams`

- {URLSearchParams}

Gets the `URLSearchParams` object representing the query parameters of the URL. This property is read-only but the `URLSearchParams` object it provides can be used to mutate the URL instance; to replace the entirety of query parameters of the URL, use the `url.search` setter. See `URLSearchParams` documentation for details.

Use care when using `.searchParams` to modify the `URL` because, per the WHATWG specification, the `URLSearchParams` object uses different rules to determine which characters to percent-encode. For instance, the `URL` object will not percent encode the ASCII tilde (`~`) character, while `URLSearchParams` will always encode it:

```
const myUrl = new URL('https://example.org/abc?foo=~bar');

console.log(myUrl.search);  // prints ?foo=~bar

// Modify the URL via searchParams...
myUrl.searchParams.sort();

console.log(myUrl.search);  // prints ?foo=%7Ebar
```

`url.username`

- {string}

Gets and sets the username portion of the URL.

```
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.username);
// Prints abc

myURL.username = '123';
console.log(myURL.href);
// Prints https://123:xyz@example.com/
```

Any invalid URL characters appearing in the value assigned the `username` property will be percent-encoded. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.toString()`

- Returns: {string}

The `toString()` method on the `URL` object returns the serialized URL. The value returned is equivalent to that of `url.href` and `url.toJSON()`.

`url.toJSON()`

- Returns: {string}

The `toJSON()` method on the `URL` object returns the serialized URL. The value returned is equivalent to that of `url.href` and `url.toString()`.

This method is automatically called when an `URL` object is serialized with `JSON.stringify()`.

```
const myURLs = [
  new URL('https://www.example.com'),
  new URL('https://test.example.org'),
];
console.log(JSON.stringify(myURLs));
// Prints ["https://www.example.com/","https://test.example.org/"]
```

`URL.createObjectURL(blob)`

> *Stability: 1 - Experimental*

- `blob` {Blob}
- Returns: {string}

Creates a `'blob:nodedata:...'` URL string that represents the given {Blob} object and can be used to retrieve the `Blob` later.

```
const {
  Blob,
  resolveObjectURL,
} = require('buffer');

const blob = new Blob(['hello']);
```

```js
const id = URL.createObjectURL(blob);

// later...

const otherBlob = resolveObjectURL(id);
console.log(otherBlob.size);
```

The data stored by the registered {Blob} will be retained in memory until `URL.revokeObjectURL()` is called to remove it.

`Blob` objects are registered within the current thread. If using Worker Threads, `Blob` objects registered within one Worker will not be available to other workers or the main thread.

`URL.revokeObjectURL(id)`

> *Stability: 1 - Experimental*

- `id` {string} A `'blob:nodedata:...` URL string returned by a prior call to `URL.createObjectURL()` .

Removes the stored {Blob} identified by the given ID. Attempting to revoke a ID that isn't registered will silently fail.

Class: `URLSearchParams`

The `URLSearchParams` API provides read and write access to the query of a `URL` . The `URLSearchParams` class can also be used standalone with one of the four following constructors. The `URLSearchParams` class is also available on the global object.

The WHATWG `URLSearchParams` interface and the `querystring` module have similar purpose, but the purpose of the `querystring` module is more general, as it allows the customization of delimiter characters (`&` and `=`). On the other hand, this API is designed purely for URL query strings.

```js
const myURL = new URL('https://example.org/?abc=123');
console.log(myURL.searchParams.get('abc'));
// Prints 123

myURL.searchParams.append('abc', 'xyz');
console.log(myURL.href);
// Prints https://example.org/?abc=123&abc=xyz

myURL.searchParams.delete('abc');
myURL.searchParams.set('a', 'b');
console.log(myURL.href);
// Prints https://example.org/?a=b

const newSearchParams = new URLSearchParams(myURL.searchParams);
// The above is equivalent to
// const newSearchParams = new URLSearchParams(myURL.search);

newSearchParams.append('a', 'c');
console.log(myURL.href);
// Prints https://example.org/?a=b
```

```js
console.log(newSearchParams.toString());
// Prints a=b&a=c

// newSearchParams.toString() is implicitly called
myURL.search = newSearchParams;
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
newSearchParams.delete('a');
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
```

`new URLSearchParams()`

Instantiate a new empty `URLSearchParams` object.

`new URLSearchParams(string)`

- `string` {string} A query string

Parse the `string` as a query string, and use it to instantiate a new `URLSearchParams` object. A leading `'?'`, if present, is ignored.

```js
let params;

params = new URLSearchParams('user=abc&query=xyz');
console.log(params.get('user'));
// Prints 'abc'
console.log(params.toString());
// Prints 'user=abc&query=xyz'

params = new URLSearchParams('?user=abc&query=xyz');
console.log(params.toString());
// Prints 'user=abc&query=xyz'
```

`new URLSearchParams(obj)`

- `obj` {Object} An object representing a collection of key-value pairs

Instantiate a new `URLSearchParams` object with a query hash map. The key and value of each property of `obj` are always coerced to strings.

Unlike [`querystring`] module, duplicate keys in the form of array values are not allowed. Arrays are stringified using [`array.toString()`], which simply joins all array elements with commas.

```js
const params = new URLSearchParams({
  user: 'abc',
  query: ['first', 'second']
});
console.log(params.getAll('query'));
// Prints [ 'first,second' ]
console.log(params.toString());
// Prints 'user=abc&query=first%2Csecond'
```

`new URLSearchParams(iterable)`

- `iterable` {Iterable} An iterable object whose elements are key-value pairs

Instantiate a new `URLSearchParams` object with an iterable map in a way that is similar to [`Map`]'s constructor. `iterable` can be an `Array` or any iterable object. That means `iterable` can be another `URLSearchParams`, in which case the constructor will simply create a clone of the provided `URLSearchParams`. Elements of `iterable` are key-value pairs, and can themselves be any iterable object.

Duplicate keys are allowed.

```js
let params;

// Using an array
params = new URLSearchParams([
  ['user', 'abc'],
  ['query', 'first'],
  ['query', 'second'],
]);
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Using a Map object
const map = new Map();
map.set('user', 'abc');
map.set('query', 'xyz');
params = new URLSearchParams(map);
console.log(params.toString());
// Prints 'user=abc&query=xyz'

// Using a generator function
function* getQueryPairs() {
  yield ['user', 'abc'];
  yield ['query', 'first'];
  yield ['query', 'second'];
}
params = new URLSearchParams(getQueryPairs());
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Each key-value pair must have exactly two elements
new URLSearchParams([
  ['user', 'abc', 'error'],
]);
// Throws TypeError [ERR_INVALID_TUPLE]:
//        Each query pair must be an iterable [name, value] tuple
```

`urlSearchParams.append(name, value)`

- `name` {string}
- `value` {string}

Append a new name-value pair to the query string.

`urlSearchParams.delete(name)`

- `name` {string}

Remove all name-value pairs whose name is `name` .

`urlSearchParams.entries()`

- Returns: {Iterator}

Returns an ES6 `Iterator` over each of the name-value pairs in the query. Each item of the iterator is a JavaScript `Array` . The first item of the `Array` is the `name` , the second item of the `Array` is the `value` .

Alias for [`urlSearchParams[@@iterator]()`](#) .

`urlSearchParams.forEach(fn[, thisArg])`

- `fn` {Function} Invoked for each name-value pair in the query
- `thisArg` {Object} To be used as `this` value for when `fn` is called

Iterates over each name-value pair in the query and invokes the given function.

```
const myURL = new URL('https://example.org/?a=b&c=d');
myURL.searchParams.forEach((value, name, searchParams) => {
  console.log(name, value, myURL.searchParams === searchParams);
});
// Prints:
//   a b true
//   c d true
```

`urlSearchParams.get(name)`

- `name` {string}
- Returns: {string} or `null` if there is no name-value pair with the given `name` .

Returns the value of the first name-value pair whose name is `name` . If there are no such pairs, `null` is returned.

`urlSearchParams.getAll(name)`

- `name` {string}
- Returns: {string[]}

Returns the values of all name-value pairs whose name is `name` . If there are no such pairs, an empty array is returned.

`urlSearchParams.has(name)`

- `name` {string}
- Returns: {boolean}

Returns `true` if there is at least one name-value pair whose name is `name` .

`urlSearchParams.keys()`

- Returns: {Iterator}

Returns an ES6 `Iterator` over the names of each name-value pair.

```js
const params = new URLSearchParams('foo=bar&foo=baz');
for (const name of params.keys()) {
  console.log(name);
}
// Prints:
//   foo
//   foo
```

`urlSearchParams.set(name, value)`

- `name` {string}
- `value` {string}

Sets the value in the `URLSearchParams` object associated with `name` to `value`. If there are any pre-existing name-value pairs whose names are `name`, set the first such pair's value to `value` and remove all others. If not, append the name-value pair to the query string.

```js
const params = new URLSearchParams();
params.append('foo', 'bar');
params.append('foo', 'baz');
params.append('abc', 'def');
console.log(params.toString());
// Prints foo=bar&foo=baz&abc=def

params.set('foo', 'def');
params.set('xyz', 'opq');
console.log(params.toString());
// Prints foo=def&abc=def&xyz=opq
```

`urlSearchParams.sort()`

Sort all existing name-value pairs in-place by their names. Sorting is done with a stable sorting algorithm, so relative order between name-value pairs with the same name is preserved.

This method can be used, in particular, to increase cache hits.

```js
const params = new URLSearchParams('query[]=abc&type=search&query[]=123');
params.sort();
console.log(params.toString());
// Prints query%5B%5D=abc&query%5B%5D=123&type=search
```

`urlSearchParams.toString()`

- Returns: {string}

Returns the search parameters serialized as a string, with characters percent-encoded where necessary.

`urlSearchParams.values()`

- Returns: {Iterator}

Returns an ES6 `Iterator` over the values of each name-value pair.

`urlSearchParams[Symbol.iterator]()`

- Returns: {Iterator}

Returns an ES6 `Iterator` over each of the name-value pairs in the query string. Each item of the iterator is a JavaScript `Array`. The first item of the `Array` is the `name`, the second item of the `Array` is the `value`.

Alias for `urlSearchParams.entries()`.

```
const params = new URLSearchParams('foo=bar&xyz=baz');
for (const [name, value] of params) {
  console.log(name, value);
}
// Prints:
//   foo bar
//   xyz baz
```

`url.domainToASCII(domain)`

- `domain` {string}
- Returns: {string}

Returns the Punycode ASCII serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to `url.domainToUnicode()`.

This feature is only available if the `node` executable was compiled with ICU enabled. If not, the domain names are passed through unchanged.

```
import url from 'url';

console.log(url.domainToASCII('español.com'));
// Prints xn--espaol-zwa.com
console.log(url.domainToASCII('中文.com'));
// Prints xn--fiq228c.com
console.log(url.domainToASCII('xn--iñvalid.com'));
// Prints an empty string
```

```
const url = require('url');

console.log(url.domainToASCII('español.com'));
// Prints xn--espaol-zwa.com
console.log(url.domainToASCII('中文.com'));
// Prints xn--fiq228c.com
console.log(url.domainToASCII('xn--iñvalid.com'));
// Prints an empty string
```

`url.domainToUnicode(domain)`

- `domain` {string}
- Returns: {string}

Returns the Unicode serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to `url.domainToASCII()`.

This feature is only available if the `node` executable was compiled with ICU enabled. If not, the domain names are passed through unchanged.

```
import url from 'url';

console.log(url.domainToUnicode('xn--espaol-zwa.com'));
// Prints español.com
console.log(url.domainToUnicode('xn--fiq228c.com'));
// Prints 中文.com
console.log(url.domainToUnicode('xn--iñvalid.com'));
// Prints an empty string
```

```
const url = require('url');

console.log(url.domainToUnicode('xn--espaol-zwa.com'));
// Prints español.com
console.log(url.domainToUnicode('xn--fiq228c.com'));
// Prints 中文.com
console.log(url.domainToUnicode('xn--iñvalid.com'));
// Prints an empty string
```

`url.fileURLToPath(url)`

- `url` {URL | string} The file URL string or URL object to convert to a path.
- Returns: {string} The fully-resolved platform-specific Node.js file path.

This function ensures the correct decodings of percent-encoded characters as well as ensuring a cross-platform valid absolute path string.

```
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);

new URL('file:///C:/path/').pathname;      // Incorrect: /C:/path/
fileURLToPath('file:///C:/path/');         // Correct:   C:\path\ (Windows)

new URL('file://nas/foo.txt').pathname;    // Incorrect: /foo.txt
fileURLToPath('file://nas/foo.txt');       // Correct:   \\nas\foo.txt (Windows)

new URL('file:///你好.txt').pathname;      // Incorrect: /%E4%BD%A0%E5%A5%BD.txt
fileURLToPath('file:///你好.txt');         // Correct:   /你好.txt (POSIX)

new URL('file:///hello world').pathname;   // Incorrect: /hello%20world
fileURLToPath('file:///hello world');      // Correct:   /hello world (POSIX)
```

```
const { fileURLToPath } = require('url');
new URL('file:///C:/path/').pathname;      // Incorrect: /C:/path/
fileURLToPath('file:///C:/path/');         // Correct:   C:\path\ (Windows)

new URL('file://nas/foo.txt').pathname;    // Incorrect: /foo.txt
fileURLToPath('file://nas/foo.txt');       // Correct:   \\nas\foo.txt (Windows)

new URL('file:///你好.txt').pathname;      // Incorrect: /%E4%BD%A0%E5%A5%BD.txt
fileURLToPath('file:///你好.txt');         // Correct:   /你好.txt (POSIX)

new URL('file:///hello world').pathname;   // Incorrect: /hello%20world
fileURLToPath('file:///hello world');      // Correct:   /hello world (POSIX)
```

`url.format(URL[, options])`

- `URL` {URL} A WHATWG URL object
- `options` {Object}
  - `auth` {boolean} `true` if the serialized URL string should include the username and password, `false` otherwise. **Default:** `true` .
  - `fragment` {boolean} `true` if the serialized URL string should include the fragment, `false` otherwise. **Default:** `true` .
  - `search` {boolean} `true` if the serialized URL string should include the search query, `false` otherwise. **Default:** `true` .
  - `unicode` {boolean} `true` if Unicode characters appearing in the host component of the URL string should be encoded directly as opposed to being Punycode encoded. **Default:** `false` .
- Returns: {string}

Returns a customizable serialization of a URL `String` representation of a WHATWG URL object.

The URL object has both a `toString()` method and `href` property that return string serializations of the URL. These are not, however, customizable in any way. The `url.format(URL[, options])` method allows for basic customization of the output.

```
import url from 'url';
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(myURL.href);
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(myURL.toString());
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(url.format(myURL, { fragment: false, unicode: true, auth: false }));
// Prints 'https://測試/?abc'
```

```
const url = require('url');
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(myURL.href);
```

```
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(myURL.toString());
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(url.format(myURL, { fragment: false, unicode: true, auth: false }));
// Prints 'https://測試/?abc'
```

`url.pathToFileURL(path)`

- `path` {string} The path to convert to a File URL.
- Returns: {URL} The file URL object.

This function ensures that `path` is resolved absolutely, and that the URL control characters are correctly encoded when converting into a File URL.

```
import { pathToFileURL } from 'url';

new URL('/foo#1', 'file:');           // Incorrect: file:///foo#1
pathToFileURL('/foo#1');              // Correct:   file:///foo%231 (POSIX)

new URL('/some/path%.c', 'file:');    // Incorrect: file:///some/path%.c
pathToFileURL('/some/path%.c');       // Correct:   file:///some/path%25.c (POSIX)
```

```
const { pathToFileURL } = require('url');
new URL(__filename);                  // Incorrect: throws (POSIX)
new URL(__filename);                  // Incorrect: C:\... (Windows)
pathToFileURL(__filename);            // Correct:   file:///... (POSIX)
pathToFileURL(__filename);            // Correct:   file:///C:/... (Windows)

new URL('/foo#1', 'file:');           // Incorrect: file:///foo#1
pathToFileURL('/foo#1');              // Correct:   file:///foo%231 (POSIX)

new URL('/some/path%.c', 'file:');    // Incorrect: file:///some/path%.c
pathToFileURL('/some/path%.c');       // Correct:   file:///some/path%25.c (POSIX)
```

`url.urlToHttpOptions(url)`

- `url` {URL} The WHATWG URL object to convert to an options object.
- Returns: {Object} Options object
  - `protocol` {string} Protocol to use.
  - `hostname` {string} A domain name or IP address of the server to issue the request to.
  - `hash` {string} The fragment portion of the URL.
  - `search` {string} The serialized query portion of the URL.
  - `pathname` {string} The path portion of the URL.
  - `path` {string} Request path. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future.
  - `href` {string} The serialized URL.

- `port` {number} Port of remote server.
- `auth` {string} Basic authentication i.e. `'user:password'` to compute an Authorization header.

This utility function converts a URL object into an ordinary options object as expected by the `http.request()` and `https.request()` APIs.

```
import { urlToHttpOptions } from 'url';
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(urlToHttpOptions(myURL));
/*
{
  protocol: 'https:',
  hostname: 'xn--g6w251d',
  hash: '#foo',
  search: '?abc',
  pathname: '/',
  path: '/?abc',
  href: 'https://a:b@xn--g6w251d/?abc#foo',
  auth: 'a:b'
}
*/
```

```
const { urlToHttpOptions } = require('url');
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(urlToHttpOptions(myUrl));
/*
{
  protocol: 'https:',
  hostname: 'xn--g6w251d',
  hash: '#foo',
  search: '?abc',
  pathname: '/',
  path: '/?abc',
  href: 'https://a:b@xn--g6w251d/?abc#foo',
  auth: 'a:b'
}
*/
```

Legacy URL API

> *Stability: 3 - Legacy: Use the WHATWG URL API instead.*

Legacy `urlObject`

> *Stability: 3 - Legacy: Use the WHATWG URL API instead.*

The legacy `urlObject` (`require('url').Url` or `import { Url } from 'url'`) is created and returned by the `url.parse()` function.

`urlObject.auth`

The `auth` property is the username and password portion of the URL, also referred to as *userinfo*. This string subset follows the `protocol` and double slashes (if present) and precedes the `host` component, delimited by `@` . The string is either the username, or it is the username and password separated by `:` .

For example: `'user:pass'` .

`urlObject.hash`

The `hash` property is the fragment identifier portion of the URL including the leading `#` character.

For example: `'#hash'` .

`urlObject.host`

The `host` property is the full lower-cased host portion of the URL, including the `port` if specified.

For example: `'sub.example.com:8080'` .

`urlObject.hostname`

The `hostname` property is the lower-cased host name portion of the `host` component *without* the `port` included.

For example: `'sub.example.com'` .

`urlObject.href`

The `href` property is the full URL string that was parsed with both the `protocol` and `host` components converted to lower-case.

For example: `'http://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'` .

`urlObject.path`

The `path` property is a concatenation of the `pathname` and `search` components.

For example: `'/p/a/t/h?query=string'` .

No decoding of the `path` is performed.

`urlObject.pathname`

The `pathname` property consists of the entire path section of the URL. This is everything following the `host` (including the `port`) and before the start of the `query` or `hash` components, delimited by either the ASCII question mark (`?`) or hash (`#`) characters.

For example: `'/p/a/t/h'` .

No decoding of the path string is performed.

`urlObject.port`

The `port` property is the numeric port portion of the `host` component.

For example: `'8080'`.

`urlObject.protocol`

The `protocol` property identifies the URL's lower-cased protocol scheme.

For example: `'http:'`.

`urlObject.query`

The `query` property is either the query string without the leading ASCII question mark (`?`), or an object returned by the querystring module's `parse()` method. Whether the `query` property is a string or object is determined by the `parseQueryString` argument passed to `url.parse()`.

For example: `'query=string'` or `{'query': 'string'}`.

If returned as a string, no decoding of the query string is performed. If returned as an object, both keys and values are decoded.

`urlObject.search`

The `search` property consists of the entire "query string" portion of the URL, including the leading ASCII question mark (`?`) character.

For example: `'?query=string'`.

No decoding of the query string is performed.

`urlObject.slashes`

The `slashes` property is a `boolean` with a value of `true` if two ASCII forward-slash characters (`/`) are required following the colon in the `protocol`.

`url.format(urlObject)`

> *Stability: 3 - Legacy: Use the WHATWG URL API instead.*

- `urlObject` {Object|string} A URL object (as returned by `url.parse()` or constructed otherwise). If a string, it is converted to an object by passing it to `url.parse()`.

The `url.format()` method returns a formatted URL string derived from `urlObject`.

```
const url = require('url');
url.format({
  protocol: 'https',
  hostname: 'example.com',
  pathname: '/some/path',
  query: {
    page: 1,
    format: 'json'
  }
});
```

```
// => 'https://example.com/some/path?page=1&format=json'
```

If `urlObject` is not an object or a string, `url.format()` will throw a `TypeError`.

The formatting process operates as follows:

- A new empty string `result` is created.
- If `urlObject.protocol` is a string, it is appended as-is to `result`.
- Otherwise, if `urlObject.protocol` is not `undefined` and is not a string, an `Error` is thrown.
- For all string values of `urlObject.protocol` that *do not end* with an ASCII colon (`:`) character, the literal string `:` will be appended to `result`.
- If either of the following conditions is true, then the literal string `//` will be appended to `result`:
    - `urlObject.slashes` property is true;
    - `urlObject.protocol` begins with `http`, `https`, `ftp`, `gopher`, or `file`;
- If the value of the `urlObject.auth` property is truthy, and either `urlObject.host` or `urlObject.hostname` are not `undefined`, the value of `urlObject.auth` will be coerced into a string and appended to `result` followed by the literal string `@`.
- If the `urlObject.host` property is `undefined` then:
    - If the `urlObject.hostname` is a string, it is appended to `result`.
    - Otherwise, if `urlObject.hostname` is not `undefined` and is not a string, an `Error` is thrown.
    - If the `urlObject.port` property value is truthy, and `urlObject.hostname` is not `undefined`:
        - The literal string `:` is appended to `result`, and
        - The value of `urlObject.port` is coerced to a string and appended to `result`.
- Otherwise, if the `urlObject.host` property value is truthy, the value of `urlObject.host` is coerced to a string and appended to `result`.
- If the `urlObject.pathname` property is a string that is not an empty string:
    - If the `urlObject.pathname` *does not start* with an ASCII forward slash (`/`), then the literal string `'/'` is appended to `result`.
    - The value of `urlObject.pathname` is appended to `result`.
- Otherwise, if `urlObject.pathname` is not `undefined` and is not a string, an `Error` is thrown.
- If the `urlObject.search` property is `undefined` and if the `urlObject.query` property is an `Object`, the literal string `?` is appended to `result` followed by the output of calling the `querystring` module's `stringify()` method passing the value of `urlObject.query`.
- Otherwise, if `urlObject.search` is a string:
    - If the value of `urlObject.search` *does not start* with the ASCII question mark (`?`) character, the literal string `?` is appended to `result`.
    - The value of `urlObject.search` is appended to `result`.
- Otherwise, if `urlObject.search` is not `undefined` and is not a string, an `Error` is thrown.
- If the `urlObject.hash` property is a string:
    - If the value of `urlObject.hash` *does not start* with the ASCII hash (`#`) character, the literal string `#` is appended to `result`.
    - The value of `urlObject.hash` is appended to `result`.

- Otherwise, if the `urlObject.hash` property is not `undefined` and is not a string, an `Error` is thrown.
- `result` is returned.

`url.parse(urlString[, parseQueryString[, slashesDenoteHost]])`

> *Stability: 3 - Legacy: Use the WHATWG URL API instead.*

- `urlString` {string} The URL string to parse.
- `parseQueryString` {boolean} If `true`, the `query` property will always be set to an object returned by the `querystring` module's `parse()` method. If `false`, the `query` property on the returned URL object will be an unparsed, undecoded string. **Default:** `false`.
- `slashesDenoteHost` {boolean} If `true`, the first token after the literal string `//` and preceding the next `/` will be interpreted as the `host`. For instance, given `//foo/bar`, the result would be `{host: 'foo', pathname: '/bar'}` rather than `{pathname: '//foo/bar'}`. **Default:** `false`.

The `url.parse()` method takes a URL string, parses it, and returns a URL object.

A `TypeError` is thrown if `urlString` is not a string.

A `URIError` is thrown if the `auth` property is present but cannot be decoded.

`url.parse()` uses a lenient, non-standard algorithm for parsing URL strings. It is prone to security issues such as host name spoofing and incorrect handling of usernames and passwords.

`url.parse()` is an exception to most of the legacy APIs. Despite its security concerns, it is legacy and not deprecated because it is:

- Faster than the alternative WHATWG `URL` parser.
- Easier to use with regards to relative URLs than the alternative WHATWG `URL` API.
- Widely relied upon within the npm ecosystem.

Use with caution.

`url.resolve(from, to)`

> *Stability: 3 - Legacy: Use the WHATWG URL API instead.*

- `from` {string} The base URL to use if `to` is a relative URL.
- `to` {string} The target URL to resolve.

The `url.resolve()` method resolves a target URL relative to a base URL in a manner similar to that of a web browser resolving an anchor tag.

```
const url = require('url');
url.resolve('/one/two/three', 'four');         // '/one/two/four'
url.resolve('http://example.com/', '/one');    // 'http://example.com/one'
url.resolve('http://example.com/one', '/two'); // 'http://example.com/two'
```

To achieve the same result using the WHATWG URL API:

```
function resolve(from, to) {
  const resolvedUrl = new URL(to, new URL(from, 'resolve://'));
  if (resolvedUrl.protocol === 'resolve:') {
    // `from` is a relative URL.
    const { pathname, search, hash } = resolvedUrl;
    return pathname + search + hash;
  }
  return resolvedUrl.toString();
}

resolve('/one/two/three', 'four');         // '/one/two/four'
resolve('http://example.com/', '/one');    // 'http://example.com/one'
resolve('http://example.com/one', '/two'); // 'http://example.com/two'
```

Percent-encoding in URLs

URLs are permitted to only contain a certain range of characters. Any character falling outside of that range must be encoded. How such characters are encoded, and which characters to encode depends entirely on where the character is located within the structure of the URL.

Legacy API

Within the Legacy API, spaces (`' '`) and the following characters will be automatically escaped in the properties of URL objects:

```
< > " ` \r \n \t { } | \ ^ '
```

For example, the ASCII space character (`' '`) is encoded as `%20`. The ASCII forward slash (`/`) character is encoded as `%3C`.

WHATWG API

The WHATWG URL Standard uses a more selective and fine grained approach to selecting encoded characters than that used by the Legacy API.

The WHATWG algorithm defines four "percent-encode sets" that describe ranges of characters that must be percent-encoded:

- The *C0 control percent-encode set* includes code points in range U+0000 to U+001F (inclusive) and all code points greater than U+007E.
- The *fragment percent-encode set* includes the *C0 control percent-encode set* and code points U+0020, U+0022, U+003C, U+003E, and U+0060.
- The *path percent-encode set* includes the *C0 control percent-encode set* and code points U+0020, U+0022, U+0023, U+003C, U+003E, U+003F, U+0060, U+007B, and U+007D.
- The *userinfo encode set* includes the *path percent-encode set* and code points U+002F, U+003A, U+003B, U+003D, U+0040, U+005B, U+005C, U+005D, U+005E, and U+007C.

The *userinfo percent-encode set* is used exclusively for username and passwords encoded within the URL. The *path percent-encode set* is used for the path of most URLs. The *fragment percent-encode set* is used for URL fragments. The *C0 control percent-encode set* is used for host and path under certain specific conditions, in addition to all other cases.

When non-ASCII characters appear within a host name, the host name is encoded using the Punycode algorithm. Note, however, that a host name *may* contain *both* Punycode encoded and percent-encoded characters:

```
const myURL = new URL('https://%CF%80.example.com/foo');
console.log(myURL.href);
// Prints https://xn--1xa.example.com/foo
console.log(myURL.origin);
// Prints https://xn--1xa.example.com
```