DNS

Stability: 2 - Stable

The `dns` module enables name resolution. For example, use it to look up IP addresses of host names.

Although named for the Domain Name System (DNS), it does not always use the DNS protocol for lookups. `dns.lookup()` uses the operating system facilities to perform name resolution. It may not need to perform any network communication. To perform name resolution the way other applications on the same system do, use `dns.lookup()`.

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

dns.lookup('example.org', (err, address, family) => {
  console.log('address: %j family: IPv%s', address, family);
});
// address: "93.184.216.34" family: IPv4
```

All other functions in the `dns` module connect to an actual DNS server to perform name resolution. They will always use the network to perform DNS queries. These functions do not use the same set of configuration files used by `dns.lookup()` (e.g. `/etc/hosts`). Use these functions to always perform DNS queries, bypassing other name-resolution facilities.

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

dns.resolve4('archive.org', (err, addresses) => {
  if (err) throw err;

  console.log(`addresses: ${JSON.stringify(addresses)}`);

  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames) => {
      if (err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```

See the Implementation considerations section for more information.

Class: `dns.Resolver`

An independent resolver for DNS requests.

Creating a new resolver uses the default server settings. Setting the servers used for a resolver using `resolver.setServers()` does not affect other resolvers:

```
const { Resolver } = require('dns');
const resolver = new Resolver();
resolver.setServers(['4.4.4.4']);

// This request will use the server at 4.4.4.4, independent of global settings.
resolver.resolve4('example.org', (err, addresses) => {
  // ...
});
```

The following methods from the `dns` module are available:

- `resolver.getServers()`
- `resolver.resolve()`
- `resolver.resolve4()`
- `resolver.resolve6()`
- `resolver.resolveAny()`
- `resolver.resolveCaa()`
- `resolver.resolveCname()`
- `resolver.resolveMx()`
- `resolver.resolveNaptr()`
- `resolver.resolveNs()`
- `resolver.resolvePtr()`
- `resolver.resolveSoa()`
- `resolver.resolveSrv()`
- `resolver.resolveTxt()`
- `resolver.reverse()`
- `resolver.setServers()`

`Resolver([options])`

Create a new resolver.

- `options` {Object}
  - `timeout` {integer} Query timeout in milliseconds, or `-1` to use the default timeout.
  - `tries` {integer} The number of tries the resolver will try contacting each name server before giving up. **Default:** `4`

`resolver.cancel()`

Cancel all outstanding DNS queries made by this resolver. The corresponding callbacks will be called with an error with code `ECANCELLED`.

`resolver.setLocalAddress([ipv4][, ipv6])`

- `ipv4` {string} A string representation of an IPv4 address. **Default:** `'0.0.0.0'`
- `ipv6` {string} A string representation of an IPv6 address. **Default:** `'::0'`

The resolver instance will send its requests from the specified IP address. This allows programs to specify outbound interfaces when used on multi-homed systems.

If a v4 or v6 address is not specified, it is set to the default, and the operating system will choose a local address automatically.

The resolver will use the v4 local address when making requests to IPv4 DNS servers, and the v6 local address when making requests to IPv6 DNS servers. The `rrtype` of resolution requests has no impact on the local address used.

`dns.getServers()`

- Returns: {string[]}

Returns an array of IP address strings, formatted according to RFC 5952, that are currently configured for DNS resolution. A string will include a port section if a custom port is used.

```
[
  '4.4.4.4',
  '2001:4860:4860::8888',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]
```

`dns.lookup(hostname[, options], callback)`

- `hostname` {string}
- `options` {integer | Object}
  - `family` {integer} The record family. Must be `4`, `6`, or `0`. The value `0` indicates that IPv4 and IPv6 addresses are both returned. **Default:** `0`.
  - `hints` {number} One or more supported `getaddrinfo` flags. Multiple flags may be passed by bitwise `OR`ing their values.
  - `all` {boolean} When `true`, the callback returns all resolved addresses in an array. Otherwise, returns a single address. **Default:** `false`.
  - `verbatim` {boolean} When `true`, the callback receives IPv4 and IPv6 addresses in the order the DNS resolver returned them. When `false`, IPv4 addresses are placed before IPv6 addresses. **Default:** `true` (addresses are not reordered). Default value is configurable using `dns.setDefaultResultOrder()` or `--dns-result-order`.
- `callback` {Function}

- `err` {Error}
- `address` {string} A string representation of an IPv4 or IPv6 address.
- `family` {integer} `4` or `6`, denoting the family of `address`, or `0` if the address is not an IPv4 or IPv6 address. `0` is a likely indicator of a bug in the name resolution service used by the operating system.

Resolves a host name (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. All `option` properties are optional. If `options` is an integer, then it must be `4` or `6` – if `options` is not provided, then IPv4 and IPv6 addresses are both returned if found.

With the `all` option set to `true`, the arguments for `callback` change to `(err, addresses)`, with `addresses` being an array of objects with the properties `address` and `family`.

On error, `err` is an `Error` object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOTFOUND'` not only when the host name does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dns.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the Implementation considerations section before using `dns.lookup()`.

Example usage:

```
const dns = require('dns');
const options = {
  family: 6,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
};
dns.lookup('example.com', options, (err, address, family) =>
  console.log('address: %j family: IPv%s', address, family));
// address: "2606:2800:220:1:248:1893:25c8:1946" family: IPv6

// When options.all is true, the result will be an Array.
options.all = true;
dns.lookup('example.com', options, (err, addresses) =>
  console.log('addresses: %j', addresses));
// addresses: [{"address":"2606:2800:220:1:248:1893:25c8:1946","family":6}]
```

If this method is invoked as its `util.promisify()`ed version, and `all` is not set to `true`, it returns a `Promise` for an `Object` with `address` and `family` properties.

Supported getaddrinfo flags

The following flags can be passed as hints to `dns.lookup()`.

- `dns.ADDRCONFIG`: Limits returned address types to the types of non-loopback addresses configured on the system. For example, IPv4 addresses are only returned if the current system has at least one IPv4 address configured.
- `dns.V4MAPPED`: If the IPv6 family was specified, but no IPv6 addresses were found, then return IPv4 mapped IPv6 addresses. It is not supported on some operating systems (e.g FreeBSD 10.1).
- `dns.ALL`: If `dns.V4MAPPED` is specified, return resolved IPv6 addresses as well as IPv4 mapped IPv6 addresses.

`dns.lookupService(address, port, callback)`

- `address` {string}
- `port` {number}
- `callback` {Function}
  - `err` {Error}
  - `hostname` {string} e.g. `example.com`
  - `service` {string} e.g. `http`

Resolves the given `address` and `port` into a host name and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

On an error, `err` is an `Error` object, where `err.code` is the error code.

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
  // Prints: localhost ssh
});
```

If this method is invoked as its `util.promisify()`ed version, it returns a `Promise` for an `Object` with `hostname` and `service` properties.

`dns.resolve(hostname[, rrtype], callback)`

- `hostname` {string} Host name to resolve.
- `rrtype` {string} Resource record type. **Default:** `'A'`.
- `callback` {Function}
  - `err` {Error}
  - `records` {string[] | Object[] | Object}

Uses the DNS protocol to resolve a host name (e.g. `'nodejs.org'`) into an array of the resource records. The `callback` function has arguments `(err, records)`.

When successful, `records` will be an array of resource records. The type and structure of individual results varies based on `rrtype`:

`rrtype`	`records` contains	Result type	Shorthand method
`'A'`	IPv4 addresses (default)	{string}	`dns.resolve4()`
`'AAAA'`	IPv6 addresses	{string}	`dns.resolve6()`
`'ANY'`	any records	{Object}	`dns.resolveAny()`
`'CAA'`	CA authorization records	{Object}	`dns.resolveCaa()`
`'CNAME'`	canonical name records	{string}	`dns.resolveCname()`
`'MX'`	mail exchange records	{Object}	`dns.resolveMx()`
`'NAPTR'`	name authority pointer records	{Object}	`dns.resolveNaptr()`
`'NS'`	name server records	{string}	`dns.resolveNs()`
`'PTR'`	pointer records	{string}	`dns.resolvePtr()`
`'SOA'`	start of authority records	{Object}	`dns.resolveSoa()`
`'SRV'`	service records	{Object}	`dns.resolveSrv()`
`'TXT'`	text records	{string[]}	`dns.resolveTxt()`

On error, `err` is an `Error` object, where `err.code` is one of the DNS error codes.

`dns.resolve4(hostname[, options], callback)`

- `hostname` {string} Host name to resolve.
- `options` {Object}
  - `ttl` {boolean} Retrieve the Time-To-Live value (TTL) of each record. When `true`, the callback receives an array of `{ address: '1.2.3.4', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.
- `callback` {Function}
  - `err` {Error}
  - `addresses` {string[] | Object[]}

Uses the DNS protocol to resolve a IPv4 addresses (`A` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

`dns.resolve6(hostname[, options], callback)`

- `hostname` {string} Host name to resolve.
- `options` {Object}
  - `ttl` {boolean} Retrieve the Time-To-Live value (TTL) of each record. When `true`, the callback receives an array of `{ address: '0:1:2:3:4:5:6:7', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

- `callback` {Function}
  - `err` {Error}
  - `addresses` {string[] | Object[]}

Uses the DNS protocol to resolve a IPv6 addresses (`AAAA` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv6 addresses.

`dns.resolveAny(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `ret` {Object[]}

Uses the DNS protocol to resolve all records (also known as `ANY` or `*` query). The `ret` argument passed to the `callback` function will be an array containing various types of records. Each object has a property `type` that indicates the type of the current record. And depending on the `type`, additional properties will be present on the object:

Type	Properties
`'A'`	`address`/`ttl`
`'AAAA'`	`address`/`ttl`
`'CNAME'`	`value`
`'MX'`	Refer to `dns.resolveMx()`
`'NAPTR'`	Refer to `dns.resolveNaptr()`
`'NS'`	`value`
`'PTR'`	`value`
`'SOA'`	Refer to `dns.resolveSoa()`
`'SRV'`	Refer to `dns.resolveSrv()`
`'TXT'`	This type of record contains an array property called `entries` which refers to `dns.resolveTxt()`, e.g. `{ entries: ['...'], type: 'TXT' }`

Here is an example of the `ret` object passed to the callback:

```
[ { type: 'A', address: '127.0.0.1', ttl: 299 },
  { type: 'CNAME', value: 'example.com' },
  { type: 'MX', exchange: 'alt4.aspmx.l.example.com', priority: 50 },
  { type: 'NS', value: 'ns1.example.com' },
  { type: 'TXT', entries: [ 'v=spf1 include:_spf.example.com ~all' ] },
  { type: 'SOA',
    nsname: 'ns1.example.com',
    hostmaster: 'admin.example.com',
    serial: 156696742,
```

```
  refresh: 900,
  retry: 900,
  expire: 1800,
  minttl: 60 } ]
```

DNS server operators may choose not to respond to `ANY` queries. It may be better to call individual methods like `dns.resolve4()`, `dns.resolveMx()`, and so on. For more details, see RFC 8482.

`dns.resolveCname(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `addresses` {string[]}

Uses the DNS protocol to resolve `CNAME` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

`dns.resolveCaa(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `records` {Object[]}

Uses the DNS protocol to resolve `CAA` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of certification authority authorization records available for the `hostname` (e.g. `[{critical: 0, iodef: 'mailto:pki@example.com'}, {critical: 128, issue: 'pki.example.com'}]`).

`dns.resolveMx(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `addresses` {Object[]}

Uses the DNS protocol to resolve mail exchange records (`MX` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

`dns.resolveNaptr(hostname, callback)`

- `hostname` {string}

- `callback` {Function}
  - `err` {Error}
  - `addresses` {Object[]}

Uses the DNS protocol to resolve regular expression based records (`NAPTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

`dns.resolveNs(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `addresses` {string[]}

Uses the DNS protocol to resolve name server records (`NS` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of name server records available for `hostname` (e.g. `['ns1.example.com', 'ns2.example.com']`).

`dns.resolvePtr(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `addresses` {string[]}

Uses the DNS protocol to resolve pointer records (`PTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of strings containing the reply records.

`dns.resolveSoa(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `address` {Object}

Uses the DNS protocol to resolve a start of authority record (`SOA` record) for the `hostname`. The `address` argument passed to the `callback` function will be an object with the following properties:

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
  minttl: 3600
}
```

`dns.resolveSrv(hostname, callback)`

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `addresses` {Object[]}

Uses the DNS protocol to resolve service records (`SRV` records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of objects with the following properties:

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
```

```
  name: 'service.example.com'
}
```

dns.resolveTxt(hostname, callback)

- `hostname` {string}
- `callback` {Function}
  - `err` {Error}
  - `records` <string[][]>

Uses the DNS protocol to resolve text queries (`TXT` records) for the `hostname`. The `records` argument passed to the `callback` function is a two-dimensional array of the text records available for `hostname` (e.g. `[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]`). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

dns.reverse(ip, callback)

- `ip` {string}
- `callback` {Function}
  - `err` {Error}
  - `hostnames` {string[]}

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of host names.

On error, `err` is an `Error` object, where `err.code` is one of the DNS error codes.

dns.setDefaultResultOrder(order)

- `order` {string} must be `'ipv4first'` or `'verbatim'`.

Set the default value of `verbatim` in `dns.lookup()` and `dnsPromises.lookup()`. The value could be:

- `ipv4first`: sets default `verbatim` `false`.
- `verbatim`: sets default `verbatim` `true`.

The default is `ipv4first` and `dns.setDefaultResultOrder()` have higher priority than `--dns-result-order`. When using worker threads, `dns.setDefaultResultOrder()` from the main thread won't affect the default dns orders in workers.

dns.setServers(servers)

- `servers` {string[]} array of RFC 5952 formatted addresses

Sets the IP address and port of servers to be used when performing DNS resolution. The `servers` argument is an array of RFC 5952 formatted addresses. If the port is the IANA default DNS port (53) it can be omitted.

```
dns.setServers([
  '4.4.4.4',
  '[2001:4860:4860::8888]',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]);
```

An error will be thrown if an invalid address is provided.

The `dns.setServers()` method must not be called while a DNS query is in progress.

The `dns.setServers()` method affects only `dns.resolve()`, `dns.resolve*()` and `dns.reverse()` (and specifically *not* `dns.lookup()`).

This method works much like resolve.conf. That is, if attempting to resolve with the first server provided results in a `NOTFOUND` error, the `resolve()` method will *not* attempt to resolve with subsequent servers provided. Fallback DNS servers will only be used if the earlier ones time out or result in some other error.

DNS promises API

The `dns.promises` API provides an alternative set of asynchronous DNS methods that return `Promise` objects rather than using callbacks. The API is accessible via `require('dns').promises` or `require('dns/promises')`.

Class: `dnsPromises.Resolver`

An independent resolver for DNS requests.

Creating a new resolver uses the default server settings. Setting the servers used for a resolver using `resolver.setServers()` does not affect other resolvers:

```
const { Resolver } = require('dns').promises;
const resolver = new Resolver();
resolver.setServers(['4.4.4.4']);

// This request will use the server at 4.4.4.4, independent of global settings.
resolver.resolve4('example.org').then((addresses) => {
  // ...
});

// Alternatively, the same code can be written using async-await style.
(async function() {
  const addresses = await resolver.resolve4('example.org');
})();
```

The following methods from the `dnsPromises` API are available:

- `resolver.getServers()`

- `resolver.resolve()`
- `resolver.resolve4()`
- `resolver.resolve6()`
- `resolver.resolveAny()`
- `resolver.resolveCaa()`
- `resolver.resolveCname()`
- `resolver.resolveMx()`
- `resolver.resolveNaptr()`
- `resolver.resolveNs()`
- `resolver.resolvePtr()`
- `resolver.resolveSoa()`
- `resolver.resolveSrv()`
- `resolver.resolveTxt()`
- `resolver.reverse()`
- `resolver.setServers()`

`resolver.cancel()`

Cancel all outstanding DNS queries made by this resolver. The corresponding promises will be rejected with an error with code `ECANCELLED`.

`dnsPromises.getServers()`

- Returns: {string[]}

Returns an array of IP address strings, formatted according to RFC 5952, that are currently configured for DNS resolution. A string will include a port section if a custom port is used.

```
[
  '4.4.4.4',
  '2001:4860:4860::8888',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]
```

`dnsPromises.lookup(hostname[, options])`

- `hostname` {string}
- `options` {integer | Object}
  - `family` {integer} The record family. Must be `4`, `6`, or `0`. The value `0` indicates that IPv4 and IPv6 addresses are both returned. **Default:** `0`.
  - `hints` {number} One or more supported `getaddrinfo` flags. Multiple flags may be passed by bitwise `OR`ing their values.
  - `all` {boolean} When `true`, the `Promise` is resolved with all addresses in an array. Otherwise, returns a single address. **Default:** `false`.

- `verbatim` {boolean} When `true`, the `Promise` is resolved with IPv4 and IPv6 addresses in the order the DNS resolver returned them. When `false`, IPv4 addresses are placed before IPv6 addresses. **Default:** currently `false` (addresses are reordered) but this is expected to change in the not too distant future. Default value is configurable using `dns.setDefaultResultOrder()` or `--dns-result-order`. New code should use `{ verbatim: true }`.

Resolves a host name (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. All `option` properties are optional. If `options` is an integer, then it must be `4` or `6` – if `options` is not provided, then IPv4 and IPv6 addresses are both returned if found.

With the `all` option set to `true`, the `Promise` is resolved with `addresses` being an array of objects with the properties `address` and `family`.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOTFOUND'` not only when the host name does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dnsPromises.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the Implementation considerations section before using `dnsPromises.lookup()`.

Example usage:

```
const dns = require('dns');
const dnsPromises = dns.promises;
const options = {
  family: 6,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
};

dnsPromises.lookup('example.com', options).then((result) => {
  console.log('address: %j family: IPv%s', result.address, result.family);
  // address: "2606:2800:220:1:248:1893:25c8:1946" family: IPv6
});

// When options.all is true, the result will be an Array.
options.all = true;
dnsPromises.lookup('example.com', options).then((result) => {
  console.log('addresses: %j', result);
  // addresses: [{"address":"2606:2800:220:1:248:1893:25c8:1946","family":6}]
});
```

`dnsPromises.lookupService(address, port)`

- `address` {string}
- `port` {number}

Resolves the given `address` and `port` into a host name and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is the error code.

```
const dnsPromises = require('dns').promises;
dnsPromises.lookupService('127.0.0.1', 22).then((result) => {
  console.log(result.hostname, result.service);
  // Prints: localhost ssh
});
```

`dnsPromises.resolve(hostname[, rrtype])`

- `hostname` {string} Host name to resolve.
- `rrtype` {string} Resource record type. **Default:** `'A'`.

Uses the DNS protocol to resolve a host name (e.g. `'nodejs.org'`) into an array of the resource records. When successful, the `Promise` is resolved with an array of resource records. The type and structure of individual results vary based on `rrtype`:

`rrtype`	`records` contains	Result type	Shorthand method
`'A'`	IPv4 addresses (default)	{string}	`dnsPromises.resolve4()`
`'AAAA'`	IPv6 addresses	{string}	`dnsPromises.resolve6()`
`'ANY'`	any records	{Object}	`dnsPromises.resolveAny()`
`'CAA'`	CA authorization records	{Object}	`dnsPromises.resolveCaa()`
`'CNAME'`	canonical name records	{string}	`dnsPromises.resolveCname()`
`'MX'`	mail exchange records	{Object}	`dnsPromises.resolveMx()`
`'NAPTR'`	name authority pointer records	{Object}	`dnsPromises.resolveNaptr()`
`'NS'`	name server records	{string}	`dnsPromises.resolveNs()`
`'PTR'`	pointer records	{string}	`dnsPromises.resolvePtr()`
`'SOA'`	start of authority records	{Object}	`dnsPromises.resolveSoa()`
`'SRV'`	service records	{Object}	`dnsPromises.resolveSrv()`
`'TXT'`	text records	{string[]}	`dnsPromises.resolveTxt()`

On error, the `Promise` is rejected with an `Error` object, where `err.code` is one of the DNS error codes.

`dnsPromises.resolve4(hostname[, options])`

- `hostname` {string} Host name to resolve.
- `options` {Object}
  - `ttl` {boolean} Retrieve the Time-To-Live value (TTL) of each record. When `true`, the `Promise` is resolved with an array of `{ address: '1.2.3.4', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

Uses the DNS protocol to resolve IPv4 addresses (`A` records) for the `hostname`. On success, the `Promise` is resolved with an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

`dnsPromises.resolve6(hostname[, options])`

- `hostname` {string} Host name to resolve.
- `options` {Object}
  - `ttl` {boolean} Retrieve the Time-To-Live value (TTL) of each record. When `true`, the `Promise` is resolved with an array of `{ address: '0:1:2:3:4:5:6:7', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

Uses the DNS protocol to resolve IPv6 addresses (`AAAA` records) for the `hostname`. On success, the `Promise` is resolved with an array of IPv6 addresses.

`dnsPromises.resolveAny(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve all records (also known as `ANY` or `*` query). On success, the `Promise` is resolved with an array containing various types of records. Each object has a property `type` that indicates the type of the current record. And depending on the `type`, additional properties will be present on the object:

Type	Properties
`'A'`	`address/ttl`
`'AAAA'`	`address/ttl`
`'CNAME'`	`value`
`'MX'`	Refer to `dnsPromises.resolveMx()`
`'NAPTR'`	Refer to `dnsPromises.resolveNaptr()`
`'NS'`	`value`
`'PTR'`	`value`
`'SOA'`	Refer to `dnsPromises.resolveSoa()`
`'SRV'`	Refer to `dnsPromises.resolveSrv()`

Type	Properties
`'TXT'`	This type of record contains an array property called `entries` which refers to `dnsPromises.resolveTxt()`, e.g. `{ entries: ['...'], type: 'TXT' }`

Here is an example of the result object:

```
[ { type: 'A', address: '127.0.0.1', ttl: 299 },
  { type: 'CNAME', value: 'example.com' },
  { type: 'MX', exchange: 'alt4.aspmx.l.example.com', priority: 50 },
  { type: 'NS', value: 'ns1.example.com' },
  { type: 'TXT', entries: [ 'v=spf1 include:_spf.example.com ~all' ] },
  { type: 'SOA',
    nsname: 'ns1.example.com',
    hostmaster: 'admin.example.com',
    serial: 156696742,
    refresh: 900,
    retry: 900,
    expire: 1800,
    minttl: 60 } ]
```

`dnsPromises.resolveCaa(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve `CAA` records for the `hostname`. On success, the `Promise` is resolved with an array of objects containing available certification authority authorization records available for the `hostname` (e.g. `[{critical: 0, iodef: 'mailto:pki@example.com'},{critical: 128, issue: 'pki.example.com'}]`).

`dnsPromises.resolveCname(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve `CNAME` records for the `hostname`. On success, the `Promise` is resolved with an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

`dnsPromises.resolveMx(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve mail exchange records (`MX` records) for the `hostname`. On success, the `Promise` is resolved with an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

`dnsPromises.resolveNaptr(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve regular expression based records (`NAPTR` records) for the `hostname`. On success, the `Promise` is resolved with an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

`dnsPromises.resolveNs(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve name server records (`NS` records) for the `hostname`. On success, the `Promise` is resolved with an array of name server records available for `hostname` (e.g. `['ns1.example.com', 'ns2.example.com']`).

`dnsPromises.resolvePtr(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve pointer records (`PTR` records) for the `hostname`. On success, the `Promise` is resolved with an array of strings containing the reply records.

`dnsPromises.resolveSoa(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve a start of authority record (`SOA` record) for the `hostname`. On success, the `Promise` is resolved with an object with the following properties:

- `nsname`

- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
  minttl: 3600
}
```

`dnsPromises.resolveSrv(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve service records (`SRV` records) for the `hostname`. On success, the `Promise` is resolved with an array of objects with the following properties:

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
  name: 'service.example.com'
}
```

`dnsPromises.resolveTxt(hostname)`

- `hostname` {string}

Uses the DNS protocol to resolve text queries (`TXT` records) for the `hostname`. On success, the `Promise` is resolved with a two-dimensional array of the text records available for `hostname` (e.g. `[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]`). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

`dnsPromises.reverse(ip)`

- `ip` {string}

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of host names.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is one of the DNS error codes.

`dnsPromises.setDefaultResultOrder(order)`

- `order` {string} must be `'ipv4first'` or `'verbatim'`.

Set the default value of `verbatim` in `dns.lookup()` and `dnsPromises.lookup()`. The value could be:

- `ipv4first`: sets default `verbatim` `false`.
- `verbatim`: sets default `verbatim` `true`.

The default is `ipv4first` and `dnsPromises.setDefaultResultOrder()` have higher priority than `--dns-result-order`. When using worker threads, `dnsPromises.setDefaultResultOrder()` from the main thread won't affect the default dns orders in workers.

`dnsPromises.setServers(servers)`

- `servers` {string[]} array of RFC 5952 formatted addresses

Sets the IP address and port of servers to be used when performing DNS resolution. The `servers` argument is an array of RFC 5952 formatted addresses. If the port is the IANA default DNS port (53) it can be omitted.

```
dnsPromises.setServers([
  '4.4.4.4',
  '[2001:4860:4860::8888]',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]);
```

An error will be thrown if an invalid address is provided.

The `dnsPromises.setServers()` method must not be called while a DNS query is in progress.

This method works much like resolve.conf. That is, if attempting to resolve with the first server provided results in a `NOTFOUND` error, the `resolve()` method will *not* attempt to resolve with subsequent servers provided. Fallback DNS servers will only be used if the earlier ones time out or result in some other error.

Error codes

Each DNS query can return one of the following error codes:

- `dns.NODATA`: DNS server returned answer with no data.
- `dns.FORMERR`: DNS server claims query was misformatted.
- `dns.SERVFAIL`: DNS server returned general failure.
- `dns.NOTFOUND`: Domain name not found.
- `dns.NOTIMP`: DNS server does not implement requested operation.
- `dns.REFUSED`: DNS server refused query.
- `dns.BADQUERY`: Misformatted DNS query.
- `dns.BADNAME`: Misformatted host name.
- `dns.BADFAMILY`: Unsupported address family.
- `dns.BADRESP`: Misformatted DNS reply.
- `dns.CONNREFUSED`: Could not contact DNS servers.
- `dns.TIMEOUT`: Timeout while contacting DNS servers.
- `dns.EOF`: End of file.
- `dns.FILE`: Error reading file.
- `dns.NOMEM`: Out of memory.
- `dns.DESTRUCTION`: Channel is being destroyed.
- `dns.BADSTR`: Misformatted string.
- `dns.BADFLAGS`: Illegal flags specified.
- `dns.NONAME`: Given host name is not numeric.
- `dns.BADHINTS`: Illegal hints flags specified.
- `dns.NOTINITIALIZED`: c-ares library initialization not yet performed.
- `dns.LOADIPHLPAPI`: Error loading `iphlpapi.dll`.
- `dns.ADDRGETNETWORKPARAMS`: Could not find `GetNetworkParams` function.
- `dns.CANCELLED`: DNS query cancelled.

Implementation considerations

Although `dns.lookup()` and the various `dns.resolve*()/dns.reverse()` functions have the same goal of associating a network name with a network address (or vice versa), their behavior is quite different. These differences can have subtle but significant consequences on the behavior of Node.js programs.

`dns.lookup()`

Under the hood, `dns.lookup()` uses the same operating system facilities as most other programs. For instance, `dns.lookup()` will almost always resolve a given name the same way as the `ping` command. On most POSIX-like operating systems, the behavior of the `dns.lookup()` function can be modified by changing settings in nsswitch.conf(5) and/or resolv.conf(5), but changing these files will change the behavior of all other programs running on the same operating system.

Though the call to `dns.lookup()` will be asynchronous from JavaScript's perspective, it is implemented as a synchronous call to getaddrinfo(3) that runs on

libuv's threadpool. This can have surprising negative performance implications for some applications, see the `UV_THREADPOOL_SIZE` documentation for more information.

Various networking APIs will call `dns.lookup()` internally to resolve host names. If that is an issue, consider resolving the host name to an address using `dns.resolve()` and using the address instead of a host name. Also, some networking APIs (such as `socket.connect()` and `dgram.createSocket()`) allow the default resolver, `dns.lookup()`, to be replaced.

`dns.resolve()`, `dns.resolve*()` and `dns.reverse()`

These functions are implemented quite differently than `dns.lookup()`. They do not use getaddrinfo(3) and they *always* perform a DNS query on the network. This network communication is always done asynchronously, and does not use libuv's threadpool.

As a result, these functions cannot have the same negative impact on other processing that happens on libuv's threadpool that `dns.lookup()` can have.

They do not use the same set of configuration files than what `dns.lookup()` uses. For instance, *they do not use the configuration from `/etc/hosts`*.