

# Cookies: HTTP State Management

Version 8.17.0.1

Jordan Johnson <jmj@fellowhuman.com>

May 4, 2025

This library provides utilities for handling cookies as specified in RFC 6265 [RFC6265].

```
(require net/cookies)      package: net-cookies-lib
```

Provides all names exported from `net/cookies/common`, `net/cookies/server`, and `net/cookies/user-agent`.

The `net/cookies/server` and `net/cookies/user-agent` modules are designed to stand on their own, however, so for any program that is exclusively client- or server-side, it will suffice to import one of those two modules.

# 1 Cookies: Common Functionality

```
(require net/cookies/common)      package: net-cookies-lib
```

The `net/cookies/common` library contains cookie-related code common to servers and user agents.

```
(cookie-name? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a valid cookie name (represented as a string or a byte string), `#f` otherwise.

Cookie names must consist of ASCII characters. They may not contain control characters (ASCII codes 0-31 or 127) or the following “separators”:

- double quotes
- whitespace characters
- `#\@` or `#\?`
- parentheses, brackets, or curly braces
- commas, colons, or semicolons
- equals, greater-than, or less-than signs
- slashes or backslashes

```
(cookie-value? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a valid cookie value (represented as a string or byte string), `#f` otherwise.

Cookie values must consist of ASCII characters. They may not contain:

- control characters
- whitespace characters
- double-quotes, except at the beginning and end if the entire value is double-quoted
- commas
- semicolons
- backslashes

```
(path/extension-value? v) → boolean?  
v : any/c
```

Returns `#t` iff `v` is a string that can be used as the value of a “Path=” attribute, or as an additional attribute (or attribute/value pair) whose meaning is not specified by RFC6265.

```
(domain-value? v) → boolean?  
v : any/c
```

Returns `#t` iff `v` is a string that contains a (sub)domain name, as defined by RFCs 1034 (Section 3.5) [RFC1034] and 1123 (Section 2.1) [RFC1123].

## 2 Cookies and HTTP Servers

```
(require net/cookies/server)      package: net-cookies-lib
```

The `net/cookies/server` library is for handling cookies on the server side; it includes:

- a serializable `cookie` structure definition
- functions to convert a cookie structure to a string, or a value for the HTTP “Set-Cookie” response header
- functions that allow reading an HTTP “Cookie” header generated by a user agent

```
(struct cookie (name
                 value
                 expires
                 max-age
                 domain
                 path
                 secure?
                 http-only?
                 extension))
name : (and/c string? cookie-name?)
value : (and/c string? cookie-value?)
expires : (or/c date? #f)
max-age : (or/c (and/c integer? positive?) #f)
domain : (or/c domain-value? #f)
path : (or/c path/extension-value? #f)
secure? : boolean?
http-only? : boolean?
extension : (or/c path/extension-value? #f)
```

A structure type for cookies the server will send to the user agent. For client-side cookies, see `net/cookies/user-agent`. Programs using this library should construct their `cookie` structs via `make-cookie`, below.

```
(make-cookie name
             value
             [#:expires exp-date
             #:max-age max-age
             #:domain domain
             #:path path
             #:secure? secure?
             #:http-only? http-only?
             #:extension extension]) → cookie?
```

```

name : cookie-name?
value : cookie-value?
exp-date : (or/c date? #f) = #f
max-age : (or/c (and/c integer? positive?) #f) = #f
domain : (or/c domain-value? #f) = #f
path : (or/c path/extension-value? #f) = #f
secure? : boolean? = #f
http-only? : boolean? = #f
extension : (or/c path/extension-value? #f) = #f

```

Constructs a cookie for sending to a user agent. If *name* or *value* is a byte string, this procedure will convert it to a string using `bytes->string/utf-8`; programs requiring a different encoding should convert their byte strings before calling `make-cookie`.

Both *exp-date* and *max-age* are for specifying a time at which the user agent should remove the cookie from its cookie store. *exp-date* is for specifying this expiration time as a date; *max-age* is for specifying it as a number of seconds in the future. If both *exp-date* and *max-age* are given, an RFC6265-compliant user agent will disregard the *exp-date* and use the *max-age*.

*domain* indicates that the recipient should send the cookie back to the server only if the hostname in the request URI is either *domain* itself, or a host within *domain*.

*path* indicates that the recipient should send the cookie back to the server only if *path* is a prefix of the request URI's path.

*secure*, when *#t*, sets a flag telling the recipient that the cookie may only be sent if the request URI's scheme specifies a "secure" protocol (presumably HTTPS).

*http-only?*, when *#t*, sets a flag telling the recipient that the cookie may be communicated only to a server and only via HTTP or HTTPS. **This flag is important for security reasons:** Browsers provide JavaScript access to cookies (for example, via `document.cookie`), and consequently, when cookies contain sensitive data such as user session info, malicious JavaScript can compromise that data. The `HttpOnly` cookie flag, set by this keyword argument, instructs the browser not to make this cookie available to JavaScript code. **If a cookie is intended to be confidential, both *http-only?* and *secure?* should be *#t*, and all connections should use HTTPS.** (Some older browsers do not support this flag; see the OWASP page on `HttpOnly` for more info.)

```

(cookie->set-cookie-header c) → bytes?
  c : cookie?

```

Produces a byte string containing the value portion of a "Set-Cookie:" HTTP response header suitable for sending *c* to a user agent.

Example:

```
> (cookie->set-cookie-header
    (make-cookie "rememberUser" "bob" #:path "/main"))
#"rememberUser=bob; Path=/main"
```

This procedure uses `string->bytes/utf-8` to convert the cookie to bytes; for an application that needs a different encoding function, use `cookie->string` and perform the bytes conversion with that function.

```
(clear-cookie-header name
    [#:domain domain
     #:path path]) → bytes?
name : cookie-name?
domain : (or/c domain-value? #f) = #f
path : (or/c path/extension-value? #f) = #f
```

Produces a byte string containing a “Set-Cookie:” header suitable for telling a user agent to clear the cookie with the given *name*. (This is done, as per RFC6265, by sending a cookie with an expiration date in the past.)

Example:

```
> (clear-cookie-header "rememberUser" #:path "/main")
#"rememberUser=; Expires=Thu, 01 Jan 2015 00:00:00 GMT;
Path=/main"
```

```
(cookie-header->alist header) → (listof (cons/c bytes? bytes?))
header : bytes?
(cookie-header->alist header decode) → (listof (cons/c X X))
header : bytes?
decode : (-> bytes? X)
```

Given the value part of a “Cookie:” header, produces an alist of all cookie name/value mappings in the header. If a *decode* function is given, applies *decode* to each key and each value before inserting the new key-value pair into the alist. Invalid cookies will not be present in the alist.

If a key in the header has no value, then `#"`, or `(decode #"`) if *decode* is present, is used as the value.

Examples:

```
> (cookie-header->alist #"SID=31d4d96e407aad42; lang=en-US")
'((#"SID" . #"31d4d96e407aad42") (#"lang" . #"en-US"))
> (cookie-header->alist #"SID=31d4d96e407aad42; lang=en-US"
    bytes->string/utf-8)
```

```
'(("SID" . "31d4d96e407aad42") ("lang" . "en-US"))
> (cookie-header->alist #"seenIntro=; logins=3"
    (compose (lambda (s) (or (string->number s) s))
              bytes->string/utf-8))
'(("seenIntro" . "") ("logins" . 3))
```

```
(cookie->string c) → string?
c : cookie?
```

Produces a string containing the given cookie as text.

Examples:

```
> (cookie->string
    (make-cookie "usesRacket" "true"))
"usesRacket=true"
> (cookie->string
    (make-cookie "favColor" "teal"
                  #:max-age 86400
                  #:domain "example.com"
                  #:secure? #t))
"favColor=teal; Max-Age=86400; Domain=example.com; Secure"
```

### 3 Cookies and HTTP User Agents

```
(require net/cookies/user-agent)    package: net-cookies-lib
```

The `net/cookies/user-agent` library provides facilities specific to user agents' handling of cookies.

Many user agents will need only two of this library's procedures:

- `extract-and-save-cookies!`, for storing cookies
- `cookie-header`, for retrieving them and generating a "Cookie:" header

```
(struct ua-cookie (name
                    value
                    domain
                    path
                    expiration-time
                    creation-time
                    access-time
                    persistent?
                    host-only?
                    secure-only?
                    http-only?))
name : cookie-name?
value : cookie-value?
domain : domain-value?
path : path/extension-value?
expiration-time : (and/c integer? positive?)
creation-time : (and/c integer? positive?)
access-time : (and/c integer? positive?)
persistent? : boolean?
host-only? : boolean?
secure-only? : boolean?
http-only? : boolean?
```

A structure representing a cookie from a user agent's point of view.

All times are represented as the number of seconds since midnight UTC, January 1, 1970, like the values produced by `current-seconds`.

It's unlikely a client will need to construct a `ua-cookie` instance directly (except perhaps for testing); `extract-cookies` produces struct instances for all the cookies received in a server's response.



```
(cookie-expired? cookie [current-time]) → boolean?
  cookie : ua-cookie?
  current-time : integer? = (current-seconds)
```

True iff the given cookie's expiration time precedes *current-time*.

### 3.1 Cookie jars: Client storage

```
(extract-and-save-cookies! headers
                             url
                             [decode]) → void?
  headers : (or/c (listof (cons/c bytes? bytes?)) (listof bytes?))
  url : url?
  decode : (-> bytes? string?) = bytes->string/utf-8
```

Reads all cookies from any “Set-Cookie” headers present in *headers* received in an HTTP response from *url*, converts them to strings using *decode*, and stores them in the *current-cookie-jar*.

The given *headers* may be provided either as an alist mapping header names to header values, or as a raw list of bytes such as the second return value produced by [http-conn-recv!](#) in *net/http-client*. Here is an example of each:

Examples:

```
> (require net/url)
> (define site-url
  (string->url "http://test.example.com/apps/main"))
> (extract-and-save-cookies!
  '(("X-Test-Header" . #"isThisACookie=no")
    ("Set-Cookie" . #"a=b; Max-Age=2000; Path=/")
    ("Set-Cookie" . #"user=bob; Max-Age=86400; Path=/apps"))
  site-url)
> (cookie-header site-url)
#"user=bob; a=b"
> (extract-and-save-cookies!
  '("#X-Ignore-This: thisIsStillNotACookie=yes"
    #Set-Cookie: p=q; Max-Age=2000; Path="/
    #Set-Cookie: usersMom=alice; Max-Age=86400; Path=/apps")
  site-url)
> (cookie-header site-url)
#"usersMom=alice; user=bob; p=q; a=b"
```

```
(save-cookie! c [via-http?]) → void?
  c : ua-cookie?
  via-http? : boolean? = #t
```

Attempts to save a single cookie *c*, received via an HTTP API iff *via-http?*, to the *current-cookie-jar*. Per Section 5.3 of RFC 6265, the cookie will be ignored if its *http-only?* flag (or that of the cookie it would replace) is set and it wasn't received via an HTTP API.

```
(cookie-header url [encode #:filter-with ok?]) → (or/c bytes? #f)
  url : url?
  encode : (-> string? bytes?) = string->bytes/utf-8
  ok? : (-> ua-cookie? boolean?) = (lambda (x) #t)
```

Finds any unexpired cookies matching *url* in the *current-cookie-jar*, removes any for which *ok?* produces *#f*, and produces the value portion of a “Cookie:” HTTP request header. Produces *#f* if no cookies match.

Cookies with the “Secure” flag will be included in this header iff (*url-scheme url*) is “https”, unless you remove them manually using the *ok?* parameter.

Example:

```
> (cookie-header
   (string->url "http://test.example.com/home"))
#"p=q; a=b"
```

```
cookie-jar<%> : interface?
```

An interface for storing cookies received from servers. Implemented by *list-cookie-jar%*. Provides for saving cookies (imperatively) and extracting all cookies that match a given URL.

Most clients will not need to deal with this interface, and none should need to call its methods directly. (Use *cookie-header* and *extract-and-save-cookies!*, instead.) It is provided for situations in which the default *list-cookie-jar%* class will not suffice. For example, if the user agent will be storing thousands of cookies, the linear insertion time of *list-cookie-jar%* could mean that writing a *cookie-jar<%>* implementation based on hash tables, trees, or a DBMS might be a better alternative.

Programs requiring such a class should install an instance of it using the *current-cookie-jar* parameter.

```
(send a-cookie-jar save-cookie! c
                                     [via-http?]) → void?
c : ua-cookie?
via-http? : boolean? = #t
```

Saves *c* to the jar, and removes any expired cookies from the jar as well.

*via-http?* should be *#t* if the cookie was received via an HTTP API; it is for properly ignoring the cookie if the cookie's *http-only?* flag is set, or if the cookie is attempting to replace an “HTTP only” cookie already present in the jar.

```
(send a-cookie-jar save-cookies! cs
                                     [via-http?]) → void?
cs : (listof ua-cookie?)
via-http? : boolean? = #t
```

Saves each cookie in *cs* to the jar, and removes any expired cookies from the jar. See the note immediately above, for explanation of the *via-http?* flag.

```
(send a-cookie-jar cookies-matching url
                                     [secure?])
→ (listof ua-cookie?)
url : url?
secure? : boolean? = (equal? (url-scheme url) "https")
```

Produces all cookies in the jar that should be sent in the “Cookie” header for a request made to *url*. *secure?* specifies whether the cookies will be sent via a secure protocol. (If not, cookies with the “Secure” flag set should not be returned by this method.)

This method should produce its cookies in the order expected according to RFC6265:

- Cookies with longer paths are listed before cookies with shorter paths.
- Among cookies that have equal-length path fields, cookies with earlier creation-times are listed before cookies with later creation-times.

If there are multiple cookies in the jar with the same name and different domains or paths, the RFC does not specify which to send. The default `list-cookie-jar%` class's implementation of this method produces **all** cookies that match the domain and path of the given URL, in the order specified above.

```
list-cookie-jar% : class?
superclass: object%
extends: cookie-jar<%>
```

Stores cookies in a list, internally maintaining a sorted order that mirrors the sort order specified by the RFC for the “Cookie” header.

```
(current-cookie-jar) → (is-a?/c cookie-jar<%>)
(current-cookie-jar jar) → void?
  jar : (is-a?/c cookie-jar<%>)
= (new list-cookie-jar%)
```

A parameter that specifies the cookie jar to use for storing and retrieving cookies.

## 3.2 Reading the Set-Cookie header

```
(extract-cookies headers url [decode]) → (listof ua-cookie?)
  headers : (or/c (listof (cons/c bytes? bytes?))
                  (listof bytes?))
  url : url?
  decode : (-> bytes? string?) = bytes->string/utf-8
```

Given a list of all the headers received in the response to a request from the given *url*, produces a list of cookies corresponding to all the “Set-Cookie” headers present. The *decode* function is used to convert the cookie’s fields to strings.

The given *headers* may be provided either as an alist mapping header names to header values, or as a raw list of bytes such as the second return value produced by [http-conn-recv!](#) in [net/http-client](#).

This function is suitable for use with the *headers/raw* field of a [request](#) structure (from [web-server/http/request-structs](#)), or with the output of [\(extract-all-fields h\)](#), where *h* is a byte string.

```
(parse-cookie set-cookie-bytes url [decode]) → (or/c ua-cookie? #f)
  set-cookie-bytes : bytes?
  url : url?
  decode : (-> bytes? string?) = bytes->string/utf-8
```

Given a single “Set-Cookie” header’s value *set-cookie-bytes* received in response to a request from the given *url*, produces a *ua-cookie* representing the cookie received, or *#f* if *set-cookie-bytes* can’t be parsed as a cookie.

The *decode* function is used to convert the cookie’s textual fields (*name*, *value*, *domain*, and *path*) to strings.

```
(default-path url) → string?  
url : url?
```

Given a URL, produces the path that should be used for a cookie that has no “Path” attribute, as specified in Section 5.1.4 of the RFC.

```
max-cookie-seconds : (and/c integer? positive?)  
min-cookie-seconds : (and/c integer? negative?)
```

The largest and smallest integers that this user agent library will use, or be guaranteed to accept, as time measurements in seconds since midnight UTC on January 1, 1970.

```
(parse-date s) → (or/c string? #f)  
s : string?
```

Parses the given string for a date, producing `#f` if it is not possible to extract a date from the string using the algorithm specified in Section 5.1.1 of the RFC.

## 4 Acknowledgements

The server-side library is based on the original [net/cookie](#) library by Francisco Solsona <solsona@acm.org>. Many of the cookie-construction tests for this library are adapted from the [net/cookie](#) tests.

Roman Klochov <kalimehtar@mail.ru> wrote the first client-side cookie library on which this user-agent library is based. In particular, this library relies on his code for parsing dates and other cookie components.

## Bibliography

- [RFC1034] P. Mockapetris, "Domain Names - Concepts and Facilities," RFC, 1987.  
<http://tools.ietf.org/html/rfc1034.html>
- [RFC1123] R. Braden (editor), "Requirements for Internet Hosts - Application and Support," RFC, 1989.  
<http://tools.ietf.org/html/rfc1123.html>
- [RFC6265] A. Barth, "HTTP State Management Mechanism," RFC, 2011.  
<http://tools.ietf.org/html/rfc6265.html>