NETWORK PROGRAMMING

CHAPTER 4 : HTTP

CHANDAN GUPTA BHAGAT

https://me.chandanbhagat.com.np

CONTENT

- The Protocol : Keep-Alive
- HTTP Methods
- The Request Body
- Cookies : CookieManager and CookiesStore

HTTP

- HyperText Transfer Protocol
- A standard that defines how a web client talks to a server and how data is transferred from the server back to the client.
- Usually thought of as a means of transferring HTML files and the pictures embedded in them.
- HTTP is data format agnostic.
- It can be used to transfer TIFF pictures, Microsoft Word documents, Windows *.exe* files, or anything else that can be represented in bytes.
- What actually happens when you type *http://www.google.com* into the browser's address bar and press Return?

The Protocol

The Protocol

- HTTP is the standard protocol for communication between web browsers and web servers. HTTP specifies how a client and server establish a connection, how the client requests data from the server, how the server responds to that request, and finally, how the connection is closed. HTTP connections use the TCP/IP protocol for data transfer. For each request from client to server, there is a sequence of four steps:
  - The client opens a TCP connection to the server on port 80, by default; other ports may be specified in the URL.
  - The client sends a message to the server requesting the resource at a specified path. The request includes a header, and optionally (depending on the nature of the request) a blank line followed by data for the request.
  - The server sends a response to the client. The response begins with a response code, followed by a header full of metadata, a blank line, and the requested document or an error message.
  - The server closes the connection.

The Protocol

- This is the basic HTTP 1.0 procedure. In HTTP 1.1 and later, multiple requests and responses can be sent in series over a single TCP connection.
- That is, steps 2 and 3 can repeat multiple times in between steps 1 and 4. Furthermore, in HTTP 1.1, requests and responses can be sent in multiple chunks. This is more scalable.
- Each request and response has the same basic form: a header line, an HTTP header containing metadata, a blank line, and then a message body. A typical client request looks something like this:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0) Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

-

The Protocol

- GET requests like this one do not contain a message body, so the request ends with a blank line.
- The first line is called the *request line*, and includes a method, a path to a resource, and the version of HTTP. The method specifies the operation being requested. The GET method asks the server to return a representation of a resource. */index.html* is the path to the resource requested from the server. HTTP/1.1 is the version of the protocol that the client understands.
- Although the request line is all that is required, a client request usually includes other information as well in a header. Each line takes the following form:

Keyword: *Value*

- Keywords are not case sensitive. Values sometimes are and sometimes aren't. Both keywords and values should be ASCII only. If a value is too long, you can add a space or tab to the beginning of the next line and continue it.
- Lines in the header are terminated by a carriage-return linefeed pair.

The Protocol

- The first keyword in this example is User-Agent, which lets the server know what browser is being used and allows it to send files optimized for the particular browser type. The following line says that the request comes from version 2.4 of the Lynx browser:

  User-Agent: Lynx/2.4 libwww/2.1.4

- All but the oldest first-generation browsers also include a Host field specifying the server's name, which allows web servers to distinguish between different named hosts served from the same IP address:

  Host: www.cafeaulait.org

- The last keyword in this example is Accept, which tells the server the types of data the client can handle (though servers often ignore this). For example, the following line says that the client can handle four MIME media types, corresponding to HTML documents, plain text, and JPEG and GIF images:

  Accept: text/html, text/plain, image/gif, image/jpeg

-

The Protocol

- MIME types are classified at two levels: a type and a subtype. The type shows very generally what kind of data is contained: is it a picture, text, or movie? The subtype identifies the specific type of data: GIF image, JPEG image, TIFF image. For example, HTML's content type is text/html; the type is text, and the subtype is html. The content type for a JPEG image is image/jpeg; the type is image, and the subtype is jpeg. Eight top-level types have been defined:
  - text/* for human-readable words
  - image/* for pictures
  - model/* for 3D models such as VRML files
  - audio/* for sound
  - video/* for moving pictures, possibly including sound
  - application/* for binary data
  - message/* for protocol-specific envelopes such as email messages and HTTP responses
  - multipart/* for containers of multiple documents and resources

The Protocol

- The response begins with a status line, followed by a header describing the response using the same “name: value” syntax as the request header, a blank line, and the requested resource. A typical successful response looks something like this:

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-length: 115
<html>
<head>
<title>
A Sample HTML file
</title>
</head>
<body>
The rest of the document goes here
</body>
</html>
```

The Protocol : Response Code

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

The Protocol : Informational responses (100 – 199)

- An informational response indicates that the request was received and understood. It is issued on a provisional basis while request processing continues.
- It alerts the client to wait for a final response.
- The message consists only of the status line and optional header fields, and is terminated by an empty line.
- As the HTTP/1.0 standard did not define any 1xx status codes, servers must not send a 1xx response to an HTTP/1.0 compliant client except under experimental conditions.
- 100 Continue
  - The server has received the request headers and the client should proceed to send the request body.
  - Sending a large request body to a server after a request has been rejected for inappropriate headers would be inefficient.
  - To have a server check the request's headers, a client must send Expect: 100-continue as a header in its initial request and receive a 100 Continue status code in response before sending the body.

The Protocol : Informational responses (100 – 199)

- 101 Switching Protocols
  - The requester has asked the server to switch protocols and the server has agreed to do so.
- 102 Processing (WebDAV; RFC 2518)
  - A WebDAV request may contain many sub-requests involving file operations, requiring a long time to complete the request.
  - This code indicates that the server has received and is processing the request, but no response is available yet.
  - This prevents the client from timing out and assuming the request was lost.
- 103 Early Hints (RFC 8297)
  - Used to return some response headers before final HTTP message

The Protocol : Successful responses (200 – 299)

- This class of status codes indicates the action requested by the client was received, understood, and accepted.
- 200 OK
  - Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request, the response will contain an entity describing or containing the result of the action.
- 201 Created
  - The request has been fulfilled, resulting in the creation of a new resource.
- 202 Accepted
  - The request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, and may be disallowed when processing occurs.
- 203 Non-Authoritative Information (since HTTP/1.1)
  - The server is a transforming proxy (e.g. a Web accelerator) that received a 200 OK from its origin, but is returning a modified version of the origin's response.
- 204 No Content
  - The server successfully processed the request, and is not returning any content.

The Protocol : Successful responses (200 – 299)

- 205 Reset Content
  - The server successfully processed the request, asks that the requester reset its document view, and is not returning any content.
- 206 Partial Content (RFC 7233)
  - The server is delivering only part of the resource (byte serving) due to a range header sent by the client. The range header is used by HTTP clients to enable resuming of interrupted downloads, or split a download into multiple simultaneous streams.
- 207 Multi-Status (WebDAV; RFC 4918)
  - The message body that follows is by default an XML message and can contain a number of separate response codes, depending on how many sub-requests were made.
- 208 Already Reported (WebDAV; RFC 5842)
  - The members of a DAV binding have already been enumerated in a preceding part of the (multistatus) response, and are not being included again.
- 226 IM Used (RFC 3229)
  - The server has fulfilled a request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

The Protocol : Redirection messages (300 – 399)

- This class of status code indicates the client must take additional action to complete the request. Many of these status codes are used in URL redirection.
- A user agent may carry out the additional action with no user interaction only if the method used in the second request is GET or HEAD. A user agent may automatically redirect a request. A user agent should detect and intervene to prevent cyclical redirects.
- 300 Multiple Choices
  - Indicates multiple options for the resource from which the client may choose (via agent-driven content negotiation). For example, this code could be used to present multiple video format options, to list files with different filename extensions, or to suggest word-sense disambiguation.
- 301 Moved Permanently
  - This and all future requests should be directed to the given URI.
- 302 Found (Previously "Moved temporarily")
  - Tells the client to look at (browse to) another URL. The HTTP/1.0 specification (RFC 1945) required the client to perform a temporary redirect with the same method (the original describing phrase was "Moved Temporarily"), but popular browsers implemented 302 redirects by changing the method to GET. Therefore, HTTP/1.1 added status codes 303 and 307 to distinguish between the two behaviours.

The Protocol : Redirection messages (300 – 399)

- 303 See Other (since HTTP/1.1)
  - The response to the request can be found under another URI using the GET method. When received in response to a POST (or PUT/DELETE), the client should presume that the server has received the data and should issue a new GET request to the given URI.
- 304 Not Modified (RFC 7232)
  - Indicates that the resource has not been modified since the version specified by the request headers If-Modified-Since or If-None-Match. In such case, there is no need to retransmit the resource since the client still has a previously-downloaded copy.
- 305 Use Proxy (since HTTP/1.1)
  - The requested resource is available only through a proxy, the address for which is provided in the response. For security reasons, many HTTP clients (such as Mozilla Firefox and Internet Explorer) do not obey this status code.
- 306 Switch Proxy
  - No longer used. Originally meant "Subsequent requests should use the specified proxy."

The Protocol : Redirection messages (300 – 399)

- 307 Temporary Redirect (since HTTP/1.1)
  - In this case, the request should be repeated with another URI; however, future requests should still use the original URI. In contrast to how 302 was historically implemented, the request method is not allowed to be changed when reissuing the original request. For example, a POST request should be repeated using another POST request.
- 308 Permanent Redirect (RFC 7538)
  - This and all future requests should be directed to the given URI. 308 parallel the behaviour of 301, but does not allow the HTTP method to change. So, for example, submitting a form to a permanently redirected resource may continue smoothly.

The Protocol : Client error responses (400 – 499)

- This class of status code is intended for situations in which the error seems to have been caused by the client. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents should display any included entity to the user.
- 400 Bad Request
  - The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, size too large, invalid request message framing, or deceptive request routing).
- 401 Unauthorized (RFC 7235)
  - Similar to 403 Forbidden, but specifically for use when authentication is required and has failed or has not yet been provided. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource. See Basic access authentication and Digest access authentication. 401 semantically means "unauthorised", the user does not have valid authentication credentials for the target resource.
- Note: Some sites incorrectly issue HTTP 401 when an IP address is banned from the website (usually the website domain) and that specific address is refused permission to access a website.

The Protocol : Client error responses (400 – 499)

- 402 Payment Required
  - Reserved for future use. The original intention was that this code might be used as part of some form of digital cash or micropayment scheme, as proposed, for example, by GNU Taler, but that has not yet happened, and this code is not widely used. Google Developers API uses this status if a particular developer has exceeded the daily limit on requests. Sipgate uses this code if an account does not have sufficient funds to start a call. Shopify uses this code when the store has not paid their fees and is temporarily disabled. Stripe uses this code for failed payments where parameters were correct, for example blocked fraudulent payments.
- 403 Forbidden
  - The request contained valid data and was understood by the server, but the server is refusing action. This may be due to the user not having the necessary permissions for a resource or needing an account of some sort, or attempting a prohibited action (e.g. creating a duplicate record where only one is allowed). This code is also typically used if the request provided authentication by answering the WWW-Authenticate header field challenge, but the server did not accept that authentication. The request should not be repeated.

The Protocol : Client error responses (400 – 499)

- 404 Not Found
  - The requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible.
- 405 Method Not Allowed
  - A request method is not supported for the requested resource; for example, a GET request on a form that requires data to be presented via POST, or a PUT request on a read-only resource.
- 406 Not Acceptable
  - The requested resource is capable of generating only content not acceptable according to the Accept headers sent in the request.
- 407 Proxy Authentication Required (RFC 7235)
  - The client must first authenticate itself with the proxy.
- 408 Request Timeout
  - The server timed out waiting for the request. According to HTTP specifications: "The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time."

The Protocol : Client error responses (400 – 499)

- 409 Conflict
  - Indicates that the request could not be processed because of conflict in the current state of the resource, such as an edit conflict between multiple simultaneous updates.
- 410 Gone
  - Indicates that the resource requested is no longer available and will not be available again. This should be used when a resource has been intentionally removed and the resource should be purged. Upon receiving a 410 status code, the client should not request the resource in the future. Clients such as search engines should remove the resource from their indices. Most use cases do not require clients and search engines to purge the resource, and a "404 Not Found" may be used instead.
- 411 Length Required
  - The request did not specify the length of its content, which is required by the requested resource.
- 412 Precondition Failed (RFC 7232)
  - The server does not meet one of the preconditions that the requester put on the request header fields.

The Protocol : Client error responses (400 – 499)

- 413 Payload Too Large (RFC 7231)
  - The request is larger than the server is willing or able to process. Previously called "Request Entity Too Large".
- 414 URI Too Long (RFC 7231)
  - The URI provided was too long for the server to process. Often the result of too much data being encoded as a query-string of a GET request, in which case it should be converted to a POST request. Called "Request-URI Too Long" previously.
- 415 Unsupported Media Type (RFC 7231)
  - The request entity has a media type which the server or resource does not support. For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.
- 416 Range Not Satisfiable (RFC 7233)
  - The client has asked for a portion of the file (byte serving), but the server cannot supply that portion. For example, if the client asked for a part of the file that lies beyond the end of the file. Called "Requested Range Not Satisfiable" previously.
- 417 Expectation Failed
  - The server cannot meet the requirements of the Expect request-header field.

The Protocol : Client error responses (400 – 499)

- 418 I'm a teapot (RFC 2324, RFC 7168)
  - This code was defined in 1998 as one of the traditional IETF April Fools' jokes, in RFC 2324, Hyper Text Coffee Pot Control Protocol, and is not expected to be implemented by actual HTTP servers. The RFC specifies this code should be returned by teapots requested to brew coffee. This HTTP status is used as an Easter egg in some websites, such as Google.com's I'm a teapot easter egg.
- 421 Misdirected Request (RFC 7540)
  - The request was directed at a server that is not able to produce a response (for example because of connection reuse).
- 422 Unprocessable Entity (WebDAV; RFC 4918)
  - The request was well-formed but was unable to be followed due to semantic errors.
- 423 Locked (WebDAV; RFC 4918)
  - The resource that is being accessed is locked.
- 424 Failed Dependency (WebDAV; RFC 4918)
  - The request failed because it depended on another request and that request failed (e.g., a PROPPATCH).

The Protocol : Client error responses (400 – 499)

- 425 Too Early (RFC 8470)
  - Indicates that the server is unwilling to risk processing a request that might be replayed.
- 426 Upgrade Required
  - The client should switch to a different protocol such as TLS/1.3, given in the Upgrade header field.
- 428 Precondition Required (RFC 6585)
  - The origin server requires the request to be conditional. Intended to prevent the 'lost update' problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict.
- 429 Too Many Requests (RFC 6585)
  - The user has sent too many requests in a given amount of time. Intended for use with rate-limiting schemes.
- 431 Request Header Fields Too Large (RFC 6585)
  - The server is unwilling to process the request because either an individual header field, or all the header fields collectively, are too large.
- 451 Unavailable For Legal Reasons (RFC 7725)
  - A server operator has received a legal demand to deny access to a resource or to a set of resources that includes the requested resource. The code 451 was chosen as a reference to the novel Fahrenheit 451 (see the Acknowledgements in the RFC).

The Protocol : Server error responses (500 – 599)

- The server failed to fulfil a request.
- Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and indicate whether it is a temporary or permanent condition. Likewise, user agents should display any included entity to the user. These response codes are applicable to any request method.
- 500 Internal Server Error
  - A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.
- 501 Not Implemented
  - The server either does not recognize the request method, or it lacks the ability to fulfil the request. Usually this implies future availability (e.g., a new feature of a web-service API).
- 502 Bad Gateway
  - The server was acting as a gateway or proxy and received an invalid response from the upstream server.

The Protocol : Server error responses (500 – 599)

- 503 Service Unavailable
  - The server cannot handle the request (because it is overloaded or down for maintenance). Generally, this is a temporary state.
- 504 Gateway Timeout
  - The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.
- 505 HTTP Version Not Supported
  - The server does not support the HTTP protocol version used in the request.
- 506 Variant Also Negotiates (RFC 2295)
  - Transparent content negotiation for the request results in a circular reference.
- 507 Insufficient Storage (WebDAV; RFC 4918)
  - The server is unable to store the representation needed to complete the request.

The Protocol : Server error responses (500 – 599)

- 508 Loop Detected (WebDAV; RFC 5842)
  - The server detected an infinite loop while processing the request (sent instead of 208 Already Reported).
- 510 Not Extended (RFC 2774)
  - Further extensions to the request are required for the server to fulfil it.
- 511 Network Authentication Required (RFC 6585)
  - The client needs to authenticate to gain network access. Intended for use by intercepting proxies used to control access to the network (e.g., "captive portals" used to require agreement to Terms of Service before granting full Internet access via a Wi-Fi hotspot).

Keep-Alive

- HTTP 1.0 opens a new connection for each request. In practice, the time taken to open and close all the connections in a typical web session can outweigh the time taken to transmit the data, especially for sessions with many small documents.
- This is even more problematic for encrypted HTTPS connections using SSL or TLS, because the hand shake to set up a secure socket is substantially more work than setting up a regular socket.
- In HTTP 1.1 and later, the server doesn’t have to close the socket after it sends its response. It can leave it open and wait for a new request from the client on the same socket.
- Multiple requests and responses can be sent in series over a single TCP connection. However, the lockstep pattern of a client request followed by a server response remains the same.
- A client indicates that it’s willing to reuse a socket by including a Connection field in the HTTP request header with the value Keep-Alive:

Connection: Keep-Alive

Keep-Alive

- The URL class transparently supports HTTP Keep-Alive unless explicitly turned off. That is, it will reuse a socket if you connect to the same server again before the server has closed the connection. You can control Java's use of HTTP Keep-Alive with several system properties:
  - Set http.keepAlive to "true or false" to enable/disable HTTP Keep-Alive. (It isenabled by default.)
  - Set http.maxConnections to the number of sockets you're willing to hold open at one time. The default is 5.
  - Set http.keepAlive.remainingData to true to let Java clean up after abandoned connections (Java 6 or later). It is false by default.
  - Set sun.net.http.errorstream.enableBuffering to true to attempt to buffer the relatively short error streams from 400- and 500-level responses, so the connection can be freed up for reuse sooner. It is false by default.
  - Set sun.net.http.errorstream.bufferSize to the number of bytes to use for buffering error streams. The default is 4,096 bytes.
  - Set sun.net.http.errorstream.timeout to the number of milliseconds before timing out a read from the error stream. It is 300 milliseconds by default

HTTP Methods

HTTP Methods

- Communication with an HTTP server follows a request-response pattern: one stateless request followed by one stateless response. Each HTTP request has two or three parts:
  - A start line containing the HTTP method and a path to the resource on which the method should be executed
  - A header of name-value fields that provide meta-information such as authentication credentials and preferred formats to be used in the request
  - A request body containing a representation of a resource (POST and PUT only)
- There are four main HTTP methods, four verbs if you will, that identify the operations that can be performed:
  - GET
  - POST
  - PUT
  - DELETE

HTTP Methods

- If that seems like too few, especially compared to the infinite number of object-oriented methods you may be accustomed to designing programs around, that's because HTTP puts most of the emphasis on the nouns: the resources identified by URIs. The uniform interface provided by these four methods is sufficient for nearly all practical purposes.
- The GET method retrieves a representation of a resource. GET is side effect free, and can be repeated without concern if it fails. Furthermore, its output is often cached, though that can be controlled with the right headers, as you'll see shortly. In a properly architected system, GET requests can be bookmarked and prefetched without concern. For example, one should not allow a file to be deleted merely by following a link because a browser may GET all links on a page before the user asks it to. By contrast, a well-behaved browser or web spider will not POST to a link without explicit user action.
- The PUT method uploads a representation of a resource to the server at a known URL. It is not side-effect free, but it is *idempotent*. That is, it can be repeated without concern if it fails. Putting the same document in the same place on the same server twice in a row leaves the server in the same state as only putting it once.

HTTP Methods

- The DELETE method removes a resource from a specified URL. It, too, is not side-effect free, but is idempotent. If you aren't sure whether a delete request succeeded—for instance, because the socket disconnected after you sent the request but before you received a response—just send the request again. Deleting the same resource twice is not a mistake.
- The POST method is the most general method. It too uploads a representation of a resource to a server at a known URL, but it does not specify what the server is to do with the newly supplied resource. For instance, the server does not necessarily have to make that resource available at the target URL, but may instead move it to a different URL. Or the server might use the data to update the state of one or more completely different resources. POST should be used for unsafe operations that should not be repeated, such as making a purchase.
- Placing the order should send a POST because that action makes a commitment. This is why browsers ask you if you're sure when you go back to a page that uses POST. Reposting data may buy two copies of a book and charge your credit card twice.

HTTP Methods

- In addition to these four main HTTP methods, a few others are used in special circumstances. The most common such method is HEAD, which acts like a GET except it only returns the header for the resource, not the actual data. This is commonly used to check the modification date of a file, to see whether a copy stored in the local cache is still valid.
- The other two that Java supports are OPTIONS, which lets the client ask the server what it can do with a specified resource; and TRACE, which echoes back the client request for debugging purposes, especially when proxy servers are misbehaving. Different servers recognize other nonstandard methods including COPY and MOVE, but Java does not send these.

The Request Body

The Request Body

- The GET method retrieves a representation of a resource identified by a URL. The exact location of the resource you want to GET from a server is specified by the various parts of the path and query string. How different paths and query strings map to different resources is determined by the server. The URL class doesn't really care about that. As long as it knows the URL, it can download from it.
- POST and PUT are more complex. In these cases, the client supplies the representation of the resource, in addition to the path and the query string. The representation of the resource is sent in the body of the request, after the header. That is, it sends these four items in order:
  - A starter line including the method, path and query string, and HTTP version
  - An HTTP header
  - A blank line (two successive carriage return/linefeed pairs)
  - The body

The Request Body

```
POST /cgi-bin/register.pl HTTP 1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-type: application/x-www-form-urlencoded
Content-length: 54
username=Elliotte+Harold&email=elharo%40ibiblio.org
```

- The body contains an *application/x-www-form-urlencoded* data, but that's just one possibility. In general, the body can contain arbitrary bytes. However, the HTTP header should include two fields that specify the nature of the body:
  - A Content-length field that specifies how many bytes are in the body (54 in the preceding example)
  - A Content-type field that specifies the MIME media type of the bytes (*application/x-www-form-urlencoded* in the preceeding example)
-

The Request Body

- Example: A camera uploading a picture to a photo sharing site can send image/jpeg. A text editor might send text/html. It's all just bytes in the end. For example, here's a PUT request that uploads an Atom document:

```
PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 322

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
<title>The Power of Pomodoros</title>
<id>urn:uuid:101a41a6-722b-4d9b-8afb-ccfb01d77499</id>
<updated>2013-02-22T19:40:52Z</updated>
<author><name>Elliotte Harold</name></author>
<content>I hadn't paid much attention to Pomodoro...</content>
</entry >
```

Cookies : CookieManager and CookiesStore

Cookies

- Many websites use small strings of text known as *cookies* to store persistent client-side state between connections.
- Cookies are passed from server to client and back again in the HTTP headers of requests and responses. It can be used by a server to indicate session IDs, shopping cart contents, login credentials, user preferences, and more. For instance, a cookie set by an online bookstore might have the value
  ISBN=0802099912&price=$34.95 to specify a book that I've put in my shopping cart. However, more likely, the value is a meaningless string such as ATVPDKIKX0DER, which identifies a particular record in a database of some kind where the real information is kept. Usually the cookie values do not contain the data but merely point to it on the server.
- Cookies are limited to non-whitespace ASCII text, and may not contain commas or semicolons.

Cookies

- To set a cookie in a browser, the server includes a Set-Cookie header line in the HTTP header. For example, this HTTP header sets the cookie “cart” to the value “ATVPDKIKX0DER”:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: cart=ATVPDKIKX0DER
```

- If a browser makes a second request to the same server, it will send the cookie back in a Cookie line in the HTTP request header like so:

```
GET /index.html HTTP/1.1
Host: www.example.org
Cookie: cart=ATVPDKIKX0DER
Accept: text/html
```

- As long as the server doesn’t reuse cookies, this enables it to track individual users and sessions across multiple, otherwise stateless, HTTP connections.

Cookies

- Servers can set more than one cookie. For example:

```
Set-Cookie:skin=noskin
Set-Cookie:ubid-main=176-5578236-9590213
Set-Cookie:session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6Ktr
Set-Cookie:session-id-time=2082787201l
Set-Cookie:session-id=187-4969589-3049309
```

- In addition to a simple name=value pair, cookies can have several attributes that control their scope including expiration date, path, domain, port, version, and security options.
- A site can also indicate that a cookie applies within an entire subdomain, not just at the original server. For example, this request sets a user cookie for the entire *foo.example.com* domain:

```
Set-Cookie: user=elharo;Domain=.foo.example.com
```

- A server can only set cookies for domains it immediately belongs to. *www.foo.example.com* cannot set a cookie for *www.oreilly.com*, *example.com*, or *.com*, no matter how it sets the domain.

Cookies

- Websites work around this restriction by embedding an image or other content hosted on one domain in a page hosted at a second domain. The cookies set by the embedded content, not the page itself, are called *third-party cookies*. Many users block all third-party cookies, and some web browsers are starting to block them by default for privacy reasons.
- Cookies are also scoped by path, so they're returned for some directories on the server, but not all. The default scope is the original URL and any subdirectories. However, the default scope can be changed using a Path attribute in the cookie. For example, this next response sends the browser a cookie with the name "user" and the value "elharo" that applies only within the server's */restricted* subtree, not on the rest of the site:

```
Set-Cookie: user=elharo; Path=/restricted
```

- When requesting a document in the subtree */restricted* from the same server, the client echoes that cookie back. However, it does not use the cookie in other directories on the site.
- A cookie can include both a domain and a path. For instance, this cookie applies in the */restricted* path on any servers within the *example.com* domain:

```
Set-Cookie: user=elharo;Path=/restricted;Domain=.example.com
```

Cookies

- The order of the different cookie attributes doesn't matter, as long as they're all separated by semicolons and the cookie's own name and value come first. However, this isn't true when the client is sending the cookie back to the server. In this case, the path must precede the domain, like so:

```
Cookie: user=elharo; Path=/restricted;Domain=.foo.example.com
```

- A cookie can be set to expire at a certain point in time by setting the expires attribute to a date in the form Wdy, DD-Mon-YYYY HH:MM:SS GMT. Weekday and month are given as three-letter abbreviations. The rest are numeric, padded with initial zeros if necessary. In the pattern language used by java.text.SimpleDateFormat, this is E, dd-MMM-yyyy H:m:s z. For instance, this cookie expires at 3:23 P.M. on December 21, 2015:

```
Set-Cookie: user=elharo; expires=Wed, 21-Dec-2015 15:23:00 GMT
```

- The browser should remove this cookie from its cache after that date has passed. The Max-Age attribute that sets the cookie to expire after a certain number of seconds have passed instead of at a specific moment. For instance, this cookie expires one hour (3,600 seconds) after it's first set:

```
Set-Cookie: user="elharo"; Max-Age=3600
```

Cookies

- Cookies can contain sensitive information such as passwords and session keys, some cookie transactions should be secure. Most of the time this means using HTTPS instead of HTTP; but whatever it means, each cookie can have a secure attribute with no value, like so:

```
Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure
```

- Browsers are supposed to refuse to send such cookies over insecure channels. For additional security against cookie-stealing attacks like XSRF, cookies can set the HttpOnly attribute. This tells the browser to only return the cookie via HTTP and HTTPS and specifically *not* by JavaScript:

```
Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure; httponly
```

- Example

```
Set-Cookie: skin=noskin; path=/; domain=.amazon.com; expires=Fri, 03-May-2013 21:46:43 GMT
Set-Cookie: ubid-main=176-5578236-9590213; path=/; domain=.amazon.com; expires=Tue, 01-Jan-
2036 08:00:01 GMT
Set-Cookie: session-
token=Zg6afPNqbaMv2WmYFOv57zCU1O6KtrMMdskcmllbZcY4q6t0PrMywqO82PR6AgtfIJhtBABhomNUW2dITwuLfOZ
uhXILp7Toya+AvWaYJxpfY1lj4ci4cnJxiuUZTev1WV31p5bcwzRM1Cmn3QOCezNNqenhzZD8TZUnOL/9Ya; path=/;
domain=.amazon.com; expires=Thu, 28-Apr-2033 21:46:43 GMT
Set-Cookie: session-id-time=2082787201l; path=/; domain=.amazon.com; expires=Tue, 01-Jan-2036
08:00:01 GMT
Set-Cookie: session-id=187-4969589-3049309; path=/; domain=.amazon.com; expires=Tue, 01-Jan-
2036 08:00:01 GMT
```

CookieManager

- Java 5 includes an abstract java.net.CookieHandler class that defines an API for storing and retrieving cookies. However, it does not include an implementation of that abstract class, so it requires a lot of grunt work. Java 6 fleshes this out by adding a concrete java.net.CookieManager subclass of CookieHandler that you can use. However, it is not turned on by default. Before Java will store and return cookies, you need to enable it:

```
CookieManager manager = new CookieManager();
CookieHandler.setDefault(manager);
```

- If all you want is to receive cookies from sites and send them back to those sites, you're done. That's all there is to it. After installing a CookieManager with those two lines of code, Java will store any cookies sent by HTTP servers you connect to with the URL class, and will send the stored cookies back to those same servers in subsequent requests.
- However, you may wish to be a bit more careful about whose cookies you accept. You can do this by specifying a CookiePolicy. Three policies are predefined:
  - CookiePolicy.ACCEPT_ALL All cookies allowed
  - CookiePolicy.ACCEPT_NONE No cookies allowed
  - CookiePolicy.ACCEPT_ORIGINAL_SERVER Only first party cookies allowed

CookieManager

- This code fragment tells Java to block third-party cookies but accept firstparty cookies:

```
CookieManager manager = new CookieManager();
manager.setCookiePolicy(CookiePolicy.ACCEPT_ORIGINAL_SERVER);
CookieHandler.setDefault(manager);
```

- That is, it will only accept cookies for the server that you're talking to, not for any server on the Internet. If you want more fine-grained control, for instance to allow cookies from some known domains but not others, you can implement the CookiePolicy interface yourself and override the shouldAccept() method:

```
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

-

CookieManager

```
import java.net.*;

public class NoGovernmentCookies implements CookiePolicy {
    @Override
    public boolean shouldAccept(URI uri, HttpCookie cookie) {
        if (uri.getAuthority().toLowerCase().endsWith(".gov")
                || cookie.getDomain().toLowerCase().endsWith(".gov")) {
            return false;
        }
        return true;
    }
}
```

CookieStore

- It is sometimes necessary to put and get cookies locally. For instance, when an application quits, it can save the cookie store to disk and load those cookies again when it next starts up. You can retrieve the store in which the CookieManager saves its cookies with the getCookieStore() method:

```
CookieStore store = manager.getCookieStore();
```

- The CookieStore class allows you to add, remove, and list cookies so you can control the cookies that are sent outside the normal flow of HTTP requests and responses:

```
public void add(URI uri, HttpCookie cookie)
public List<HttpCookie> get(URI uri)
public List<HttpCookie> getCookies()
public List<URI> getURIs()
public boolean remove(URI uri, HttpCookie cookie)
public boolean removeAll()
```

- Each cookie in the store is encapsulated in an HttpCookie object that provides methods for inspecting the attributes of the cookie summarized

CookieStore

- It is sometimes necessary to put and get cookies locally. For instance, when an application quits, it can save the cookie store to disk and load those cookies again when it next starts up. You can retrieve the store in which the CookieManager saves its cookies with the getCookieStore() method:

```
CookieStore store = manager.getCookieStore();
```

- The CookieStore class allows you to add, remove, and list cookies so you can control the cookies that are sent outside the normal flow of HTTP requests and responses:

```
public void add(URI uri, HttpCookie cookie)
public List<HttpCookie> get(URI uri)
public List<HttpCookie> getCookies()
public List<URI> getURIs()
public boolean remove(URI uri, HttpCookie cookie)
public boolean removeAll()
```

- Each cookie in the store is encapsulated in an HttpCookie object that provides methods for inspecting the attributes of the cookie summarized

CookieStore

```
package java.net;

public class HttpCookie implements Cloneable {

    public HttpCookie(String name, String value)

    public boolean hasExpired()

    public void setComment(String comment)

    public String getComment()

    public void setCommentURL(String url)

    public String getCommentURL()

    public void setDiscard(boolean discard)

    public boolean getDiscard()

    public void setPortlist(String ports)

    public String getPortlist()

    public void setDomain(String domain)

    public String getDomain()

    public void setMaxAge(long expiry)

    public long getMaxAge()

    public void setPath(String path)

    public String getPath()

    public void setSecure(boolean flag)

    public boolean getSecure()

    public String getName()

    public void setValue(String value)

    public String getValue()

    public int getVersion()

    public void setVersion(int v)

    public static boolean domainMatches(String domain, String
host)

    public static List<HttpCookie> parse(String header)

    public String toString()

    public boolean equals(Object obj)

    public int hashCode()

    public Object clone()

}
```

THANK YOU

YOU CAN FIND THIS SLIDES/NOTES IN

https://chandanbhagat.com.np/docs/network-programming/

AND CODES IN

https://github.com/chandan-g-bhagat/network-programming