5 Web Browsers

In this chapter, we go over the fundamental considerations in designing and building a Web browser, as well as other sophisticated Web clients. When discussing Web browsers, our focus will not be on the graphical aspects of browser functionality (i.e. the layout of pages, the rendering of images). Instead, we shall concentrate on the issues associated with the processing of HTTP requests and responses. The value of this knowledge will become apparent as we proceed to our discussion of more sophisticated Web applications.

It may seem to some that the task of designing a browser is a fait accompli, a foregone conclusion, a done deal, a known problem that has already been ‘solved’. Given the history and progress of browser development—from the original *www* browser, through Lynx and Mosaic, to Netscape, Internet Explorer, and Opera today—it might seem a futile endeavor to ‘reinvent the wheel’ by building a new browser application.

This is hardly the case at all. The desktop browser is the most obvious example of a Web client, and it’s certainly the most common, but it’s far from the only one. Other types of Web clients include *agents*, which are responsible for submitting requests on behalf of a user to perform some automated function, and *proxies*, which act as gateways through which requests and responses pass between servers and clients to enhance security and performance. These clients need to replicate much of the functionality found in browsers. Thus, it is worthwhile to understand design principles associated with browser architecture.

Furthermore, there are devices like handheld *personal digital assistants*, cellular phones, and *Internet appliances*, which need to receive and send data via the Web. Although many of them have browsers available already, they are mostly primitive with limited functionality. As the capabilities of these devices grow, more advanced and robust Web clients will be needed.

Finally, who said that today’s desktop browsers are perfect examples of elegant design? The Mozilla project is an effort to build a better browser from the ground up (from the ashes of an existing one, if you will). Today's desktop browsers may be (relatively) stable, and it would be difficult if not impossible to develop and market a new desktop browser at this stage of the game. Still, there will be ample opportunities to enhance and augment the functionality of existing desktop browsers, and this effort is best undertaken with a thorough understanding of the issues of browser design.

The main responsibilities of a browser are as follows:

1. Generate and send requests to Web servers on the user's behalf, as a result of following hyperlinks, explicit typing of URLs, submitting forms, and parsing HTML pages that require auxiliary resources (e.g. images, applets).
2. Accept responses delivered by Web servers and interpret them to produce the visual representation to be viewed by the user. This will, at a bare minimum, involve examination of certain response headers such as `Content-Type` to determine what action needs to be taken and what sort of rendering is required.
3. Render the results in the browser window or through a third party tool, depending on the content type of the response.

This, of course, is an oversimplification of what real browsers actually do. Depending on the status code and headers in the response, browsers are called upon to perform other tasks, including:

1. *Caching*: the browser must make determinations as to whether or not it needs to request data from the server at all. It may have a cached copy of the same data item that it retrieved during a previous request. If so, and if this cached copy has not 'expired', the browser can eliminate a superfluous request for the resource. In other cases, the server can be queried to determine if the resource has been modified since it was originally retrieved and placed in the cache. Significant performance benefits can be achieved through caching.
2. *Authentication*: since web servers may require authorization credentials to access resources it has designated as secure, the browser must react to server requests for credentials, by prompting the user for authorization credentials, or by utilizing credentials it has already asked for in prior requests.
3. *State maintenance*: to record and maintain the state of a browser session across requests and responses, web servers may request that the browser accept *cookies*, which are sets of name/value pairs included in response headers. The browser must store the transmitted cookie information and make it available to be sent back in appropriate requests. In addition, the browser should provide configuration options to allow users the choice of accepting or rejecting cookies.
4. *Requesting supporting data items*: the typical web page contains images, Java applets, sounds, and a variety of other ancillary objects. The proper rendering

of the page is dependent upon the browser's retrieving those supporting data items for inclusion in the rendering process. This normally occurs transparently without user intervention.

5. *Taking actions in response to other headers and status codes*: the HTTP headers and the status code do more than simply provide the data to be rendered by the browser. In some cases, they provide additional processing instructions, which may extend or supersede rendering information found elsewhere in the response. The presence of these instructions may indicate a problem in accessing the resource, and may instruct the browser to *redirect* the request to another location. They may also indicate that the connection should be kept open, so that further requests can be sent over the same connection. Many of these functions are associated with advanced HTTP functionality found in HTTP/1.1.
6. *Rendering complex objects*: most web browsers inherently support content types such as `text/html`, `text/plain`, `image/gif`, and `image/jpeg`. This means that the browser provides native functionality to render objects with these contents *inline*: within the browser window, and without having to install additional software components. To render or play back other more complex objects (e.g. audio, video, and multimedia), a browser must provide support for these content types. Mechanisms must exist for invoking external *helper applications* or internal *plug-ins* that are required to display and playback these objects.
7. *Dealing with error conditions*: connection failures and invalid responses from servers are among the situations the browser must be equipped to deal with.

5.1 ARCHITECTURAL CONSIDERATIONS

So, let's engage in an intellectual exercise: putting together requirements for the architecture of a Web browser. What are those requirements? What functions must a Web browser perform? And how do different functional components interact with each other?

The following list delineates the core functions associated with a Web browser. Each function can be thought of as a distinct module within the browser. Obviously these modules must communicate with each other in order to allow the browser to function, but they should each be designed atomically.

- *User Interface*: this module is responsible for providing the interface through which users interact with the application. This includes presenting, displaying, and rendering the end result of the browser's processing of the response transmitted by the server.
- *Request Generation*: this module bears responsibility for the task of building HTTP requests to be submitted to HTTP servers. When asked by the *User*

Interface module or the *Content Interpretation* module to construct requests based on relative links, it must first resolve those links into absolute URLs.

- *Response Processing*: this module must parse the response, interpret it, and pass the result to the *User Interface* module.
- *Networking*: this module is responsible for network communications. It takes requests passed to it by the *Request Generation* module and transmits them over the network to the appropriate Web server or proxy. It also accepts responses that arrive over the network and passes them to the *Response Processing* module. In the course of performing these tasks, it takes responsibility for establishing network connections and dealing with proxy servers specified in a user's network configuration options.
- *Content Interpretation*: having received the response, the *Response Processing* module needs help in parsing and deciphering the content. The content may be encoded, and this module is responding to decode it. Initial responses often have their content types set to `text/html`, but HTML responses embed or contain references to images, multimedia objects, JavaScript code, applets, and style sheet information. This module performs the additional processing necessary for browser applications to understand these entities within a response. In addition, this module must tell the *Request Generation* module to construct additional requests for the retrieval of auxiliary content such as images, applets, and other objects.
- *Caching*: caching provides web browsers with a way to economize by avoiding the unnecessary retrieval of resources that the browser already has a usable copy of, 'cached' away in local storage. Browsers can ask Web servers whether a desired resource has been modified since the time that the browser initially retrieved it and stored it in the cache. This module must provide facilities for storing copies of retrieved resources in the cache for later use, for accessing those copies when viable, and for managing the space (both memory and disk) allocated by the browser's configuration parameters for this purpose.
- *State Maintenance*: since HTTP is a *stateless protocol*, some mechanism must be in place to maintain the browser *state* between related requests and responses. Cookies are the mechanism of choice for performing this task, and support for cookies is in the responsibility of this module.
- *Authentication*: this module takes care of composing authorization credentials when requested by the server. It must interpret response headers demanding credentials by prompting the user to enter them (usually via a dialog). It must also store those credentials, but only for the duration of the current browser session, in case a request is made for another secured resource in what the server considers to be the same security 'realm'. (This absolves the user of the need to re-enter the credentials each time a request for such resources is made.)

- *Configuration*: finally, there are a number of configuration options that a browser application needs to support. Some of these are fixed, while others are user-definable. This module maintains the fixed and variable configuration options for the browser, and provides an interface for users to modify those options under their control.

5.2 PROCESSING FLOW

Figure 5.1 shows the processing flow for the creation and transmission of a request in a typical browser. We begin with a link followed by a user. Users can click on hyperlinks presented in the browser display window, they might choose links from lists of previously visited links (history or bookmarks), or they might enter a URL manually.

In each of these cases, processing begins with the *User Interface* module, which is responsible for presenting the display window and giving users access to browser functions (e.g. through menus and shortcut keys). In general, an application using a *GUI* (graphical user interface) operates using an *event model*. User actions—clicking on highlighted hyperlinks, for example—are considered *events* that must be interpreted properly by the *User Interface* module. Although this book does not concentrate on the user interface-related functionality of HTTP browsers, it is crucial that we note the events that are important for the *User Interface* module:

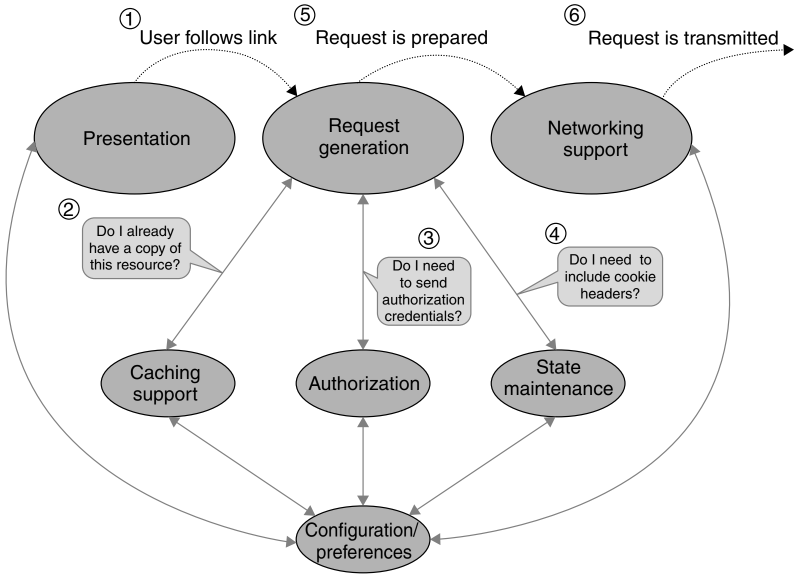


Figure 5.1 Browser request generation

- *Entering URLs manually*: usually, this is accomplished by providing a text entry box in which the user can enter a URL, as well as through a menu option (**File→ Open**) that opens a dialog box for similar manual entry. The second option often interfaces with the operating system to support interactive selection of local files.
- *Selecting previously visited links*: the existence of this mechanism, naturally, implies that the *User Interface* module must also provide a mechanism for maintaining a history of visited links. The maximum amount of time that such links will be maintained in this list, as well as the maximum size to which this list can grow, can be established as a user-definable parameter in the *Configuration* module. The 'Location' or 'Address' text area in the browser window can be a *dropdown* field that allows the user to select from recently visited links. The 'Back' button allows users to go back to the page they were visiting previously. In addition, users should be able to save particular links as "bookmarks", and then access these links through the user interface at a later date.
- *Selecting displayed hyperlinks*: there are a number of ways for users to select links displayed on the presented page. In desktop browsers, the mouse click is probably the most common mechanism for users to select a displayed link, but there are other mechanisms on the desktop and on other platforms as well. Since the *User Interface* module is already responsible for rendering text according to the specifications found in the page's HTML markup, it is also responsible for doing some sort of formatting to highlight a link so that it stands out from other text on the page. Most desktop browsers also change the cursor shape when the mouse is 'over' a hyperlink, indicating that this is a valid place for users to click. Highlighting mechanisms vary for non-desktop platforms, but they should always be present in some form.

Once the selected or entered link is passed on to the *Request Generation* module, it must be *resolved*. Links found on a displayed page can be either absolute or relative. Absolute URLs are complete URLs, containing all the required URL components, e.g. `protocol://host/path`. These do not need to be resolved and can be processed without further intervention. A relative URL specifies a location relative to:

1. the current location being displayed (i.e. the entire URL including the path, up to the directory in which the current URL resides), when the HREF contains a relative path that does not begin with a slash, e.g.: `<A HREF="some_directory/just_a_file_name.html">`), or
2. the current location's web server root (i.e., only the host portion of the URL), when the HREF contains a relative path that *does* begin with a slash, e.g. `<A HREF="/root_level_directory/another_file_name.html">`.

```
Current URL:  http://www.myserver.com/mydirectory/index.html
<A HREF ="anotherdirectory/page2.html">...</A>
        → http://www.myserver.com/mydirectory/anotherdirectory/page2.html
<A HREF ="/rootleveldirectory/homepage.html">...</A>
        → http://www.myserver.com/rootleveldirectory/homepage.html

Current URL:  http://www.myserver.com/mydirectory/anotherpage.html
<A HREF ="anotherdirectory/page2.html">...</A>
        → http://www.myserver.com/mydirectory/anotherdirectory/page2.html
<A HREF ="/yetanotherdirectory/homepage.html">...</A>
        → http://www.myserver.com/yetanotherdirectory/homepage.html

Current URL:  http://www.myserver.com/mydirectory/differentpage.html
<BASE HREF ="http://www.yourserver.com/otherdir/something.html">
<A HREF ="anotherdirectory/page2.html">...</A>
        → http://www.yourserver.com/otherdir/anotherdirectory/page2.html
<A HREF ="/yetanotherdirectory/homepage.html">...</A>
        → http://www.yourserver.com/yetanotherdirectory/homepage.html
```

Figure 5.2 Resolution of relative URLs

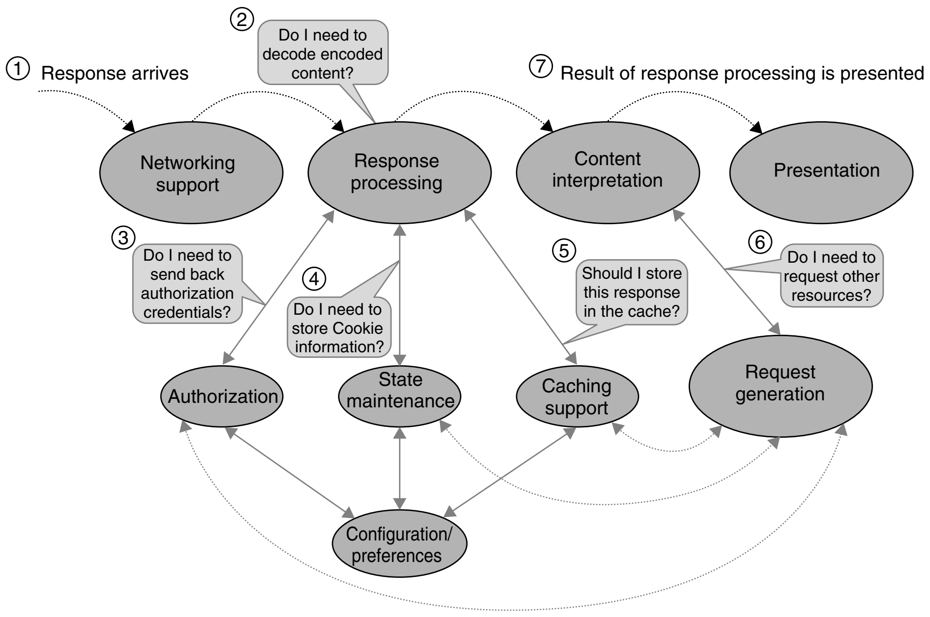
The process of resolution changes if an optional `<BASE HREF ="...">` tag is found in the `HEAD` section of the page. The URL specified in this tag replaces the current location as the "base" from which resolution occurs in the previous examples.

Figure 5.2 demonstrates how relative URLs must be resolved by the browser.

Once the URL has been resolved, the *Request Generation* module builds the request, which is ultimately passed to the *Networking* module for transmission. To accomplish this task, the *Request Generation* module has to communicate with other browser components:

- It asks the *Caching* module *"Do I already have a copy of this resource?"* If so, it needs to determine whether it can simply use this copy, or whether it needs to ask the server if the resource has been modified since the browser cached a copy of this resource.
- It asks the *Authorization* module *"Do I need to include authentication credentials in this request?"* If the browser has not already stored credentials for the appropriate domain, it may need to contact the *User Interface* module, which prompts the user for credentials.
- It asks the *State Mechanism* module *"Do I need to include* `Cookie` *headers in this request?"* It must determine whether the requested URL matches domain and path patterns associated with previously stored cookies.

The constructed request is passed to the *Networking* module so it can be transmitted over the network.



① Response arrives
② Do I need to decode encoded content?
⑦ Result of response processing is presented
Networking support
Response processing
Content interpretation
Presentation
③ Do I need to send back authorization credentials?
④ Do I need to store Cookie information?
⑤ Should I store this response in the cache?
⑥ Do I need to request other resources?
Authorization
State maintenance
Caching support
Request generation
Configuration/ preferences

Figure 5.3 Browser response processing

Once a request has been transmitted, the browser waits to receive a response. It may submit additional requests while waiting. Requests may have to be resubmitted if the connection is closed before the corresponding responses are received. It is the server's responsibility to transmit responses in the same order as the corresponding requests were received. However, the browser is responsible for dealing with servers that do not properly maintain this order, by delaying the processing of responses that arrive out of sequence. Figure 5.3 describes the flow for this process. A response is received by the *Networking* module, which passes it to the *Response Processing* module. This module must also cooperate and communicate with other modules to do its job. It examines response headers to determine required actions.

- If the status code of the response is `401 Not Authorized`, this means that the request lacked necessary authorization credentials. The *Response Processing* module asks the *Authorization* module whether any existing credentials might be used to satisfy the request. The *Authorization* module may, in turn, contact the *User Interface* module, which would prompt the user to enter authorization credentials. In either case, this results in the original request being retransmitted with an `Authorization` header containing the required credentials.
- If the response contains `Set-Cookie` headers, the *State Maintenance* module must store the cookie information using the browser's persistence mechanism.

Next, the response is passed to the *Content Interpretation* module, which has a number of responsibilities:

- If the response contains `Content-Transfer-Encoding` and/or `Content-Encoding` headers, the module needs to *decode* the body of the response.
- The module examines the `Cache-Control`, `Expires`, and/or `Pragma` headers (depending on the HTTP version of the response) to determine whether the browser needs to cache the decoded content of the response. If so, the *Caching* module is contacted to create a new cache entry or update an existing one.
- The `Content-Type` header determines the MIME type of the response content. Different MIME types, naturally, require different kinds of content processing. Modern browsers support a variety of content types natively, including HTML (`text/html`), graphical images (`image/gif`, and `image/jpeg`), and sounds (`audio/wav`). Native support means that processing of these content types is performed by built-in browser components. Thus, the *Content Interpretation* module must provide robust support for such processing. Leading edge browsers already provide support for additional content types, including vector graphics and XSL stylesheets.
- For MIME types that are not processed natively, browsers usually provide support mechanisms for the association of MIME types with *helper applications* and *plug-ins*. Helper applications render content by invoking an external program that executes independent of the browser, while plug-ins render content within the browser window. The *Content Interpretation* module must communicate with the *Configuration* module to determine what plug-ins are installed and what helper application associations have been established, to take appropriate action when receiving content that is not natively supported by the browser. This involves a degree of interaction with the operating system, to determine system-level associations configured for filename extensions, MIME types, and application programs. However, many browsers override (or even completely ignore) these settings, managing their own sets of associations through the *Configuration* module.
- Some content types (e.g. markup languages, applets, Flash movies) may embed references to other resources needed to satisfy the request. For instance, HTML pages may include references to images or JavaScript components. The *Content Interpretation* module must parse the content prior to passing it on to the *User Interface* module, determining if additional requests will be needed. If so, URLs associated with these requests get resolved when they are passed to the Request Generation module.

As each of the requested resources arrives in sequence, it is passed to the *User Interface* module so that it may be incorporated in the final presentation. The *Networking* module maintains its queue of requests and responses, ensuring that all requests have been satisfied, and resubmitting any outstanding requests.

All along the way, various subordinate modules are asked questions to determine the course of processing (including whether or not particular tasks need to be performed at all). For example, the *Content Interpretation* module may say '*This page has IMG tags, so we must send HTTP requests to retrieve the associated images,*' but the *Caching* module may respond by saying '*We already have a usable copy of that resource, so don't bother sending a request to the network for it.*' (Alternatively, it may say '*We have a copy of that resource, but let's ask the server if its copy of the resource is more recent; if it's not, it doesn't need to send it back to us.*') Or the *Configuration* module may say '*No, don't send a request for the images on this page, this user has a slow connection and has elected not to see images.*' Or *the State Maintenance* mechanism may jump in and say '*Wait, we've been to this site before, so send along this identifying cookie information with our requests.*'

The rest of this chapter is devoted to a more detailed explanation of the role each of these modules plays in the processing of requests and responses. As mentioned previously, we shall not focus on the *User Interface* module's responsibility in rendering graphics, as this is an extensive subject worthy its own book. However, we will concentrate on the interplay between these modules and how to design them to do their job.

We begin by going over the basics of request and response processing, following that with details on the more sophisticated aspects of such processing, including support for caching, authentication, and advanced features of the HTTP protocol.

5.3 PROCESSING HTTP REQUESTS AND RESPONSES

Let us examine how browsers build and transmit HTTP requests, and how they receive, interpret, and present HTTP responses. After we have covered the basics of constructing requests and interpreting responses, we can look at the more complex interactions involved when HTTP transactions involve caching, authorization, cookies, request of supporting data items, and multimedia support.

Not Just HTTP

Browsers should do more than just communicate via HTTP. They should provide support for Secure HTTP (over Secure Sockets Layer). They should be able to send requests to FTP servers. And they should be able to access local files. These three types of requests correspond to URLs using the `https`, `ftp`, and `file` protocols, respectively. Although we shall not cover these protocols here, it is important to note that HTTP requests and responses are not the only kinds of transactions performed by browsers.