WEB SERVERS AND HTTP

OBJECTIVES

- Describe the technologies that comprise the World Wide Web.
- Explain and use URLs and URIs.
- Read and understand HTTP requests, responses, and headers.
- Describe how HTML forms interact with the web server and its programs.

The World Wide Web

- The World Wide Web was invented in 1989 by programmer **Tim Berners-Lee**.
    - His project started as a way to simplify collaboration among scientists at particle physics labs around the world.
        - He quickly recognized the global value of this new Internet application.
    - He wrote the first web server in 1990, then the first HTML browser/editor (which he named *WorldWideWeb*.)
        - He published his work online in the USENET newsgroup discussion forums, the main Internet collaboration app of the time.
- He launched the first web site in August, 1991.
- Berners-Lee combined three components to create the Web:
    - The HyperText Markup Language (**HTML**) based on the much more extensive markup language SGML, and its dialect HyTime.
        - HTML defines document structure and presentation.
    - Uniform Resource Identifiers (**URI**s) and Uniform Resource Locators (**URL**s).
        - These provide a concise, consistent way of uniquely identifying and retrieving a resource on the Internet.
    - The Hypertext Transfer Protocol (**HTTP**), a protocol language for retrieving and publishing content.
        - This allows browsers and web servers to communicate in a simple but flexible and extensible way.

After writing a proposal in 1989 describing his idea, Tim Berners-Lee wrote the first web server software in 1990 in his lab at CERN, the particle-physics facility in Switzerland (where Berners-Lee worked as a physicist). He also wrote an HTML browser/editor program, and he and colleagues refined and developed his concept and his code and put together the first few web pages by the end of that year. After further testing and development, in August 1991 he put the first web site online, and published his results in USENET newsgroup *alt.hypertext.*

Interest in the World Wide Web was immediate and enthusiastic. Programmers started working on their own web servers and browsers. Programmers at the *National Centers for Supercomputing Applications* (NCSA) at the University of Illinois, Urbana-Champaign developed a web server and released it as open-source software. Would-be webmasters downloaded, compiled, and installed it to create early web sites. Other NCSA programmers, led by Marc Andreeson, created a web browser named *Mosaic*. Andreeson left NCSA to found what would become Netscape Communications and sell commercial web servers and browsers based on the NCSA work. *Netscape Navigator* was very successful and was the predominant browser through the mid-1990s until Microsoft's *Internet Explorer* (IE) began to dominate. Netscape released their browser source code, and the Mozilla open-source project took over, developing the *Mozilla* browser which then became *Firefox*, the most popular open-source browser. IE ruled the browser market until Google's *Chrome* browser overtook it in 2012.

Development of NCSA's web server, the *NCSA httpd* ("HTTP daemon") spun down. Commercial web servers (Netscape Server, Microsoft Internet Information Server) became available, but a new open-source project produced *Apache*, a web server based on the NCSA httpd. Its development was rapid and because it was free many organizations chose it over commercial alternatives. The project spawned a number of other open-source projects under the name Apache Project, with the web server named *Apache httpd*. It remained the dominant web server from the mid 1990s through at least the end of 2014.

The World Wide Web Consortium
Tim Berners-Lee went on to co-found and lead the World Wide Web Consortium (known as W3C), the international organization that oversees web-related standards including HTML, CSS, XML, DOM, SOAP, WSDL, and many others. Current versions of standards can be found at *http://www.w3.org/standards*.

HTML

- An **HTML** directive or *tag* consists of a tag name, with optional *attributes*, contained within angle brackets (<>).
  - Tags are human-readable and embedded in the text they mark up.
  - Browsers interpret the tags and render the content accordingly.
    - Browsers ignore any tags they don't understand
  - Between the tags are the text content of the page.
  - Some tags are references to non-text content to be rendered into the page.
  - All white space in text - spaces, tabs, newlines, etc. is discarded by the browser as it decides how to render the text.
- A tag is either empty (standalone) or non-empty.
  - An empty tag's presence represents the content to include.

    ```
    <hr /> <!-- Draw a horizontal rule (line) here -->
    ```

  - An non-empty tag determines how to treat the text between it and its closing tag.

    ```
    <p>This is a paragraph, to be rendered with blank lines
    above and below.</p>
    ```

- **href** anchor tags produce a clickable (or otherwise activated) hyperlink to another page or resource, thus creating hypertext.

  ```
  <a href="/content/fullstory/42.html">Full story</a>
  ```

The structure of an HTML document consists of:

```
<!DOCTYPE html>
<html>
<head>
<title>This is my web page</title>
</head>
<body>
<h1>This is the main header of my page</h1>
<hr />
<p>This    is the first  paragraph of my page content.
Extra blanks and newlines
will be discarded, and the browser will flow the text according
to the tags (and stylesheet directives, if present).
</body>
</html>
```

Early versions of HTML focused on formatting text, originally for scientific publications with titles, sections, subheads, paragraphs, quotations, lists, and the like. Tags were created for defining appearance and layout: a **`<center>`** tag instructing the browser to horizontally center everything enclosed; a **`<font>`** tag specifying basic properties (typeface, relative size, color, etc.) of the enclosed text - but separate **`<b>`** and **`<i>`** tags for bold and italic text. The syntax of HTML was somewhat loose and informal, and browsers were expected to be forgiving of things like unclosed or improperly-nested tags. HTML focused on formatting content for human readers, and all layout and styling was done with HTML tags and attributes.

As the web grew rapidly in the early 1990s and HTML evolved, new tags proliferated, each with its own attributes for controlling their specific styling options, making it harder for authors to learn and for browsers to keep up with in implementation. Web proponents saw a need to separate document presentation from document content, and out of several competing ideas **CSS** (Cascading Style Sheets) emerged. Styling attributes and tags began to disappear from HTML, with pages instead accompanied by embedded or separately referenced stylesheets that instruct the browser to select certain elements and apply style rules to them. CSS and HTML have evolved side-by-side ever since.

The forgiving nature of HTML browsers, and the resulting sloppy structure of much early web content, made it difficult to write programs to reliably and meaningfully parse and search web pages. Early on, as the web was just beginning to develop, Tim Berners-Lee and others proposed moving toward a markup, and a web, which defined not the superficial appearance of text in a browser, but which described the *meaning* of the content, and in a way easy to parse by programs as well: the *Semantic Web*. The success of this vision is debatable but out of it grew some important standards including XML (eXtensible Markup Language), a more rigidly-defined syntax that can be reliably parsed and navigated by programs.

URIS AND URLS

- A fundamental component of the web is the need to identify, in a concise and globally unique way, a particular information resource.
  - For this Tim Berners-Lee devised what became the *Uniform Resource Identifier* or URI.
- A **URI** is text string beginning with a *scheme*, followed by a colon, followed by a *scheme-specific part*.
  - What follows the first colon depends on the specific scheme.
  - One scheme, *urn* (for Uniform Resource Name), allows referencing of various existing identification systems:

```
urn:isbn:978-0-596-00357-9
urn:ietf:rfc:3896
```

- The most important kind of Internet URI is the *Uniform Resource Locator* or **URL**, which specifies how to retrieve a resource.
  - In an Internet URL, the scheme is typically an application protocol, followed by a hostname and a path to the resource on that host:
    - Login and port information can be included.

```
http://www.example.com/content/somepage.html
ftp://user2:mypass@ftp.example.com/pub/somefile.txt
rtsp://media.example.com:88/videos/somevideo.avi
```

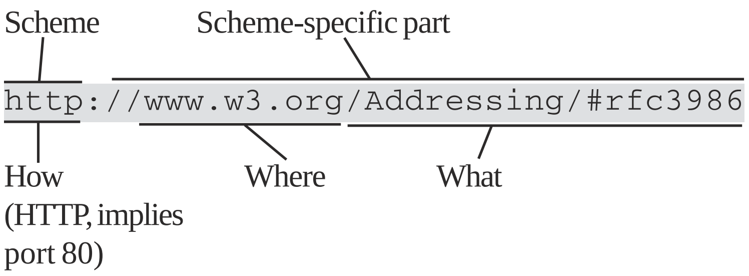
  - Other forms of URL are common:

```
mailto:dobbs@example.com
jdbc:oracle:thin:@localhost:1521/testdb
```

Internet **URI**s leverage the existing Domain Name System (DNS), which establishes registered, globally-unique, hierarchical domain names. That is, a resouce identified with the URI *http://example.com/schemas* can be assumed to be unique to, and controlled by, the owners of the *example.com* domain. Note that a URI does NOT necessarily refer to an actual item that can be retrieved. In this example, there may be no page to be found at *http://example.com/schemas*, or even a web server with that name. The URI is just used to differentiate a resource from somebody else's. This form of URI is commonly used with XML namespaces.

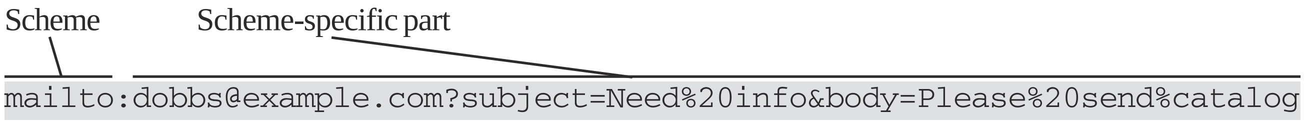
A **URL** implies there is actually a resource to be accessed at the named server using the specified scheme. URLs are the basis of HTML hypertext links. A URL provides sufficient information to access the reqource over a network. The port number is implied, defaulting to the well-known port associated with the protocol identified by the scheme. If the server is listening on an alternate port, the port number follows the hostname, separated by a colon.

Web URL:



```
http://www.w3.org/Addressing/#rfc3986
```

Mail URL:



```
mailto:dobbs@example.com?subject=Need%20info&body=Please%20send%catalog
```

HTTP: Hypertext Transfer Protocol

- The third key component of the web is the *Hypertext Trasfer Protocol*, **HTTP**.
  - This is the language a browser or other client program uses to request a page from a web server, and with which the server responds.
- HTTP is a simple protocol transmitted in human-readable text
  - It includes only a small number of possible requests: **GET**, **POST**, **HEAD**, **OPTIONS**, etc.
- A **request** consists of *REQUEST RESOURCE_URI PROTOCOL_VERSION*

```
GET /content/somepage.html HTTP/1.0
```

  - The client can optionally send request headers.

```
GET /content/somepage.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
```

- A **response** begins *PROTOCOL_VERSION RESPONSE_CODE REASON*

```
HTTP/1.1 200 OK
```

  - Optional headers can (and just about always do) follow the response line.

```
HTTP/1.1 301 Moved permanently
Content-Length: 322

<html><head><title>301 Moved permanently</title></head>
<body><p>The document has moved <a href="http://example.com/
pages/somepage.html">here</a>.</p><hr> .....
```

- An extra blank line separates the request or response and its headers from any content to follow.

URL Encoding

A URL is a text string that cannot contain spaces (spaces delimit the three parts of a valid HTTP request), or any characters other than letters, digits, `-`, `_`, `.`, and `~`. Other characters, such as `/`, `?`, `&`, `=`, and so on have special meanings in web URLs. To include these special characters as literals in a URL we must encode them using *URL encoding*:

* Each space character can be encoded as a plus sign: `+`:
  `http://example.com/My+Cool+Page.html`
  represents a resource named "`/My Cool Page.html`"

* Other reserved characters are encoded with *percent encoding*: the character is replaced by its hexidecimal character code, preceded by a percent sign:
  `http://example.com/What+Is+HTML%2FCSS%2FJavaScript%3F.html`
  Represents a resource named "`/What Is HTML/CSS/JavaScript?.html`"

Note that a space can be represented either as a `+`, or as `%20`. A newline is `%0A`. A percent sign is `%25`.

HTTP HEADERS

- HTTP defines four general categories of header:
  - General headers
  - Request headers
  - Response headers
  - Entity headers
    - In HTTP the term *entity* refers to any content sent by the server or client.
- HTTP headers follow a format used by earlier protocols such as SMTP and NNTP.

  `Header-name: [header value]`

  - The header name occurs at the beginning of a line.
    - Any following lines that begin with whitespace (space or tab) are part of the preceding header's value.
  - Header names are case-insensitive and do not contain spaces.
  - A header is not required to have a value.
- Headers allow HTTP clients and servers to:
  - Exchange "meta-information" about documents.
  - Negotiate conditional requests (`If-Modified-Since: date`).
  - Negotiate prefered and acceptible content types.
  - Perform HTTP basic or digest access authentication.
  - Control caching and proxy server behavior.

Example HTTP GET transaction:

```
                              (Client connects to web server via socket connection on port 80)
GET /index.html HTTP/1.0                                                         Request line
Connection: Keep-Alive                                                         General header
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:11.0)                       Request header
Accept: image/gif, image/x-xbitmap, image/jpeg, */*                            Request header
                                           Carriage-return terminates client request headers
HTTP/1.1 200 OK                                                                 Response line
Connection: Close                                                              General header
Date: Mon, 04 Aug 2014 20:28:06 GMT                                            General header
Server: Apache/2.2.29 (CentOS)                                                Response header
Content-Type: text/html                                                         Entity header
Content-Length: 249                                                             Entity header
                                         Carriage-return terminates server response headers
<HTML>                                                               Entity body from server
<HEAD><TITLE>Welcome to my Web Site</TITLE></HEAD>                   Entity body from server
<BODY>This is a sample page...                                       Entity body from server
                               (Server closes socket connection after the entity-body is sent)
```

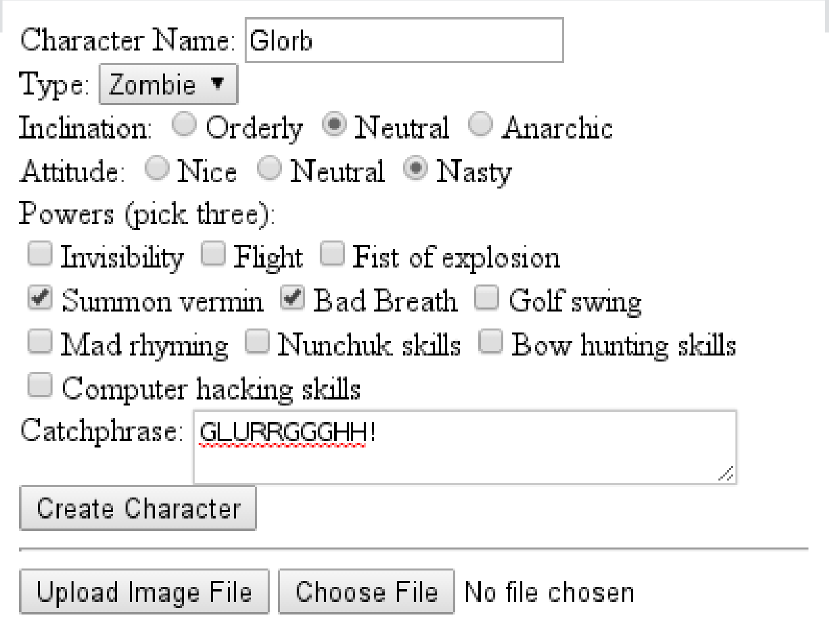
Example HTTP POST transaction:

```
                              (Client connects to web server via socket connection on port 80)
POST /cgi-bin/guestbook HTTP/1.0                                                 Request line
Connection: Keep-Alive                                                         General header
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:11.0)                       Request header
Accept: image/gif, image/x-xbitmap, image/jpeg, */*                            Request header
Content-Type: text/plain                                                        Entity header
Content-Length: 37                                                              Entity header
                                           Carriage-return terminates client request headers
Hello, I'm Bob Dobbs! Nice web site!                                 Entity body from client
                                                                     Entity body from client
HTTP/1.1 200 OK                                                                 Response line
Connection: Close                                                              General header
Date: Mon, 04 Aug 2014 20:28:06 GMT                                            General header
Server: Apache/2.2.29 (CentOS)                                                Response header
Content-Type: text/html                                                         Entity header
Content-Length: 100                                                             Entity header
                                         Carriage-return terminates server response headers
<HTML>                                                               Entity body from server
<HEAD><TITLE>Thanks!</TITLE></HEAD>                                  Entity body from server
<BODY>You've been added to my guest book.</BODY>                     Entity body from server
</HTML>                                                              Entity body from server
                               (Server closes socket connection after the entity-body is sent)
```

HTML Forms

- An *HTML form* provides a mechanism for accepting user input in a browser and, when activated, generating and sending a query request to the web server.
  - The web server typically invokes a separate program or script to handle the query and generate the HTML content of the response.
- A web page can contain any number of forms.
  - Each form is enclosed in `<form></form>` tags.
    - A form `name` attribute can help both client-side JavaScript and server-side code distinguish between forms on a page.
  - The form's **action** attribute specifies the server-side resource to handle the form's content.
- HTML form elements provide familiar user-interface components.
  - Simple inputs like text fields, buttons, drop-down lists, checkboxes, and radio button groups.
  - More complex widgets like file selectors, date and color choosers, etc.
  - Hidden elements, often used to cache data in a multi-form sequence.
- Each form element has a name and a value.
  - An element can have multiple values.
  - The browser sends each element's name and value to the server when the form is activated.
    - The server passes these *name=value* pairs to the CGI script designated by the form's action.

```
<form name='character_detail' method='POST' action='http://example.com/char.cgi'>
 <label>Character Name:<input type="text" name='character_name'></label><br />
 <label>Type: <select name='character_type'>
                <option value="pir">Pirate</option>
                <option value="nin">Ninja</option>
                <option value="zom">Zombie</option>
                <option value="rob">Robot</option>
              </select></label><br />
 <label>Inclination:
 <input type='radio' name='inclination' value='orderly'>Orderly</input>
 <input type='radio' name='inclination' value='neutral'>Neutral</input>
 <input type='radio' name='inclination' value='anarchic'>Anarchic</input></label>
 <br /><label>Attitude:
 <input type='radio' name='attitude' value='nice'>Nice</input>
 <input type='radio' name='attitude' value='neutral'>Neutral</input>
 <input type='radio' name='attitude' value='nasty'>Nasty</input></label>
 <br /><label>Powers (pick three):<br />
  <input type='checkbox' name='power' value='Invisibility'>Invisibility</input>
...
  <input type='checkbox' name='power' value='Computer hacking skills'>
                                    Computer hacking skills</input></label><br />
  <label>Catchphrase:
  <textarea name='catchphrase' rows='2' cols='30' placeholder='Character 
catchphrase, eg: "Imma sort this mess OUT!"' style='vertical-align:top'></
textarea></label><br />
  <input type='hidden' name='player_level' value='1' />
 <button name='create_character' value='Create'>Create Character</button> <br />
</form><hr />
<form name='avatar_upload' method='POST' action='http://example.com/char.cgi'
      enctype="multipart/form-data">
  <button name='upload_avatar' value='Upload'>Upload Image File</button>
  <input type='file' name='avatarfile' accept='image/*' /><br />
</form>
```


Character Name: Glorb
Type: Zombie
Inclination: ○ Orderly ● Neutral ○ Anarchic
Attitude: ○ Nice ○ Neutral ● Nasty
Powers (pick three):
☐ Invisibility ☐ Flight ☐ Fist of explosion
☑ Summon vermin ☑ Bad Breath ☐ Golf swing
☐ Mad rhyming ☐ Nunchuk skills ☐ Bow hunting skills
☐ Computer hacking skills
Catchphrase: GLURRGGGHH!

HTML FORM ACTIVATION

- The two most important form tag attributes are:
  - `action="`*`URI`*`"` - the name of the server-side program that was written to handle this particular form's input.
  - `method="`*`{`*`GET`*`|`*`POST`*`}`*`"` - whether to use the HTTP **GET** method or the HTTP **POST** method to transmit the form content.
- An HTML form needs an element that will *activate* it, causing the browser to transmit the user's input to the server.
  - The most obvious and traditional example is a **Submit** button.
  - A form's first **Submit** button is its *default button*.
    - Hitting *Enter* in the browser triggers the default button.
- Activating the form causes the browser to:
  - Concatenate the form element names and current values into a properly-encoded CGI *query string*.
  - Send the specified HTTP **method** request to the specified **action** URL, along with the query string.

```
POST /char.cgi HTTP/1.1
Host: example.com
Connection: keep-alive
Content-Length: 179
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,*/*;q=0.8
Origin: null
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/40.0.2214.91 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8

character_name=Glorb&character_type=Zombie&inclination=neutral&attitude=
nasty&power=Summon+vermin&power=Bad+Breath&catchphrase=GLURRGGGHH%21&pla
yer_level=1&create_character=Create

HTTP/1.1 200 Ok
Connection: close
Content-type: text/plain
Content-length: 18

Character created.
```

HTTP Response Codes

- HTTP server response codes are three-digit numbers and categorized, with each category using a range of 100 possible codes.

1xx Informational.

- *Request received, continuing response*. Seldom used.

2xx Successful.

- `200 OK` is the standard success response for a **GET** or **POST**.
- `201 Created` request resulted in creation of a resource on the server.
  - The server should return the URL representing the new resource in the **Location:** response header..
- `206 Partial Content`: client sent **GET** with a `Range:` request header, server provided just the requested range of content.

3xx Redirection.

- Resource has moved, or has multiple versions, etc.; repeat request with the new URI provided in the response.

4xx Client Error.

- `404 Not found` is common - client requested a nonexistent resource.
- `401 Unauthorized` initiates an HTTP authentication exchange.
- `403 Forbidden` might indicate a security violation.

5xx Server Error.

- `500 Internal Server Error` usually indicates a problem with server-side code - a syntax error in a PHP or Perl script for example, or a database error.
- `501 Not Implemented`: client sent an invalid request command.
- `503 Service Unavailable`: temporary server overload.

Class	Category	Code	Recommended Reason Phrase
1*xx*	Informational - Request received, continuing process	100 101	Continue Switching Protocols
2*xx*	Success - The action was successfully received, understood, and accepted	200 201 202 203 204 205 206	OK Created Accepted Non-Authoritative Information No Content Reset Content Partial Content
3*xx*	Redirection - Further action must be taken in order to complet the request	300 301 302 303 304 305	Multiple Choices Moved Permanently Moved Temporarily See Other Not Modified Use Proxy
4*xx*	Client Error - The request contains bad syntax or cannot be fulfilled	400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415	Bad Request Unauthorized Payment Required Forbidden Not Found Method Not Allowed Not Acceptible Proxy Authentication Required Request Time-out Conflict Gone Length Required Precondition Failed Request Entity Too Large Request URI Too Large Unsupported Media Type
5*xx*	Server Error - The server failed to fulfill an apparently valid request	500 501 502 503 504 505	Internal Server Error Not Implemented Bad Gateway Service Unavailable Gateway Time-out HTTP Version not supported

Labs

❶ Use **telnet** to connect to port 80 of *www.google.com* and do an HTTP/1.0 GET request for the main page. What value was returned in the *Content-type:* header?

❷ Telnet to port 80 of *www.google.com* and do a proper HTTP/1.1 GET request for the main page.

❸ Using **telnet** connections to port 80 of *www.google.com*, try the following requests noting the HTTP response line and headers returned for each:

```
HEAD / HTTP/1.0
OPTIONS / HTTP/1.0
TRACE / HTTP/1.0
```

Try these requests for some other web sites.

❹ Open *newcharacter.html* in a text editor. For both forms, change the **action** attribute to *'mailto:me@example.com'* (feel free to substitute your own actual email address, though it's not necessary.) Load the page in your browser, fill out the form text inputs and select various other items, then click the "*Create Character*" button.

When a form has a *mailto:* **action** URL, most browsers will open your default email client program and pass the form data to it. Examine what was loaded into the email program. You can close the email message window when you're done (if you used your own email address in the mailto: URL, feel free to send it, and examine the resulting message when it arrives.)

❺ Edit *newcharacter.html* again, modifying the action URL to: *'mailto:me@example.com?subject="HTTP+Parameter+Demo'*. Reload the page in your browser and fill out the form. In the "Catchprase" input add some text that includes spaces, question marks, ampersands, slashes, and so on. Submit the form and look closely at the request body that loads into your email program. Close the email message window when you're done.

❻ Open your browser's **Developer Tools**:

*	Chrome: *Ctrl-Shift-I* (*Cmd-Option-I* on Mac), or Menu | More Tools | Developer tools
*	Firefox: *Ctrl-Shift-K* (*Cmd-Option-K* on Mac), or Tools | Web Developer | Web Console
*	Safari: *Cmd-Option-I*, or Preferences | Advanced | Show Develop Menu, then Develop | Show Web Inspector,
*	Internet Explorer: *F12* (Developer Tools) | Console

Now modify newcharacter.html again, changing the **action** back to *'http://example.com/char.cgi'*. Fill out the form and submit it again. This will result in a *404 - Not Found* error from *example.com* (which of course has no such script online to process our form). Find the Network tab in the Developer Tools. Select the POST request for *char.cgi*, and explore the various HTTP headers as well as the form's query parameters.