Week 7, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Announcements

- CS50 Lunch is again this Friday 10/24 at 1:15pm, RSVP at the usual http://cs50.harvard.edu/rsvp.
- Jason dressed as a pumpkin one year for his section that happened to land on Halloween 2011 (Section 8[1] at cs50.tv[2]), and that year his air pump was working, but by 2012 his costume was deflated.
- If you want to join us in pumpkin carving with Daven and Gabe this Friday 10/24, 3pm, RSVP by emailing heads@cs50.harvard.edu[3].
- The final project will be discussed soon, but they can be almost any project of interest to you, with the approval of your teaching fellow.
- To help you with that, we will have seminars, optional classes taught by teaching fellows and other staff from across campus, on various topics that are fun and different. This year's lineup:

[1] http://cs50.tv/2011/fall/sections/8/section8.mp4
[2] http://cs50.tv
[3] mailto:heads@cs50.harvard.edu

    # 3D Modeling and Manufacture
    # Amazing Web Apps with Ruby on Rails
    # Android 101
    # Breaking Through The (Google) Glass Ceiling
    # Build Tomorrow's Library
    # Cloud Computing with Amazon Web Services (AWS)
    # CSS: Awesome Style and Design
    # Data Analysis in R
    # Data Visualization and D3
    # Essential Scale-Out Computing
    # Exposing Digital Photography
    # How to Build Innovative Technologies
    # iOS App Development with Swift
    # Learning iOS: Create your own app with Objective-C!
    # Light Your World (with Hue Bulbs)
    # Meteor: a better way to build apps
    # Teach Your Computer to See: Augmented Reality with OpenCV
    # Transitioning to Agile Development from Waterfall

- If you want to register or see more information about these topics, visit http://cs50.harvard.edu/register.

HTTP Review

- We began and concluded on Monday with HTTP, Hypertext Transfer Protocol, which is just the way your web browser speaks to a web server, like the way humans extend a hand to each other to make a handshake.
- A protocol is just a set of conventions of sending information back and forth, like this familiar picture:
- The most common method used is **GET**, that looks like this:

```
GET / HTTP/1.1
Host: www.google.com
...
```

- And that request is placed inside a digital envelope with the IP address of the computer it is from, and the IP address of the recipient. But we need one more piece of information, the port number, since the server might be listening for different kinds of communication. Port 80 is the most common as it's the default for web traffic, so we'll see that often, along with 443, used for secure web traffic (HTTPS).
- Also remember that the `/` after `GET` means that we want the root of the web server, or the default webpage.
- Hopefully the server responds with something like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

- And the number 200 is a convention saying that everything okay, with the `Content-Type` of the message `text/html`, and the `…` will be the webpage itself.
- So now we can pick up by writing HTML, Hypertext Markup Language, that we'll use to write webpages and specify their structure and style.

HTML

- On Monday we looked at this:

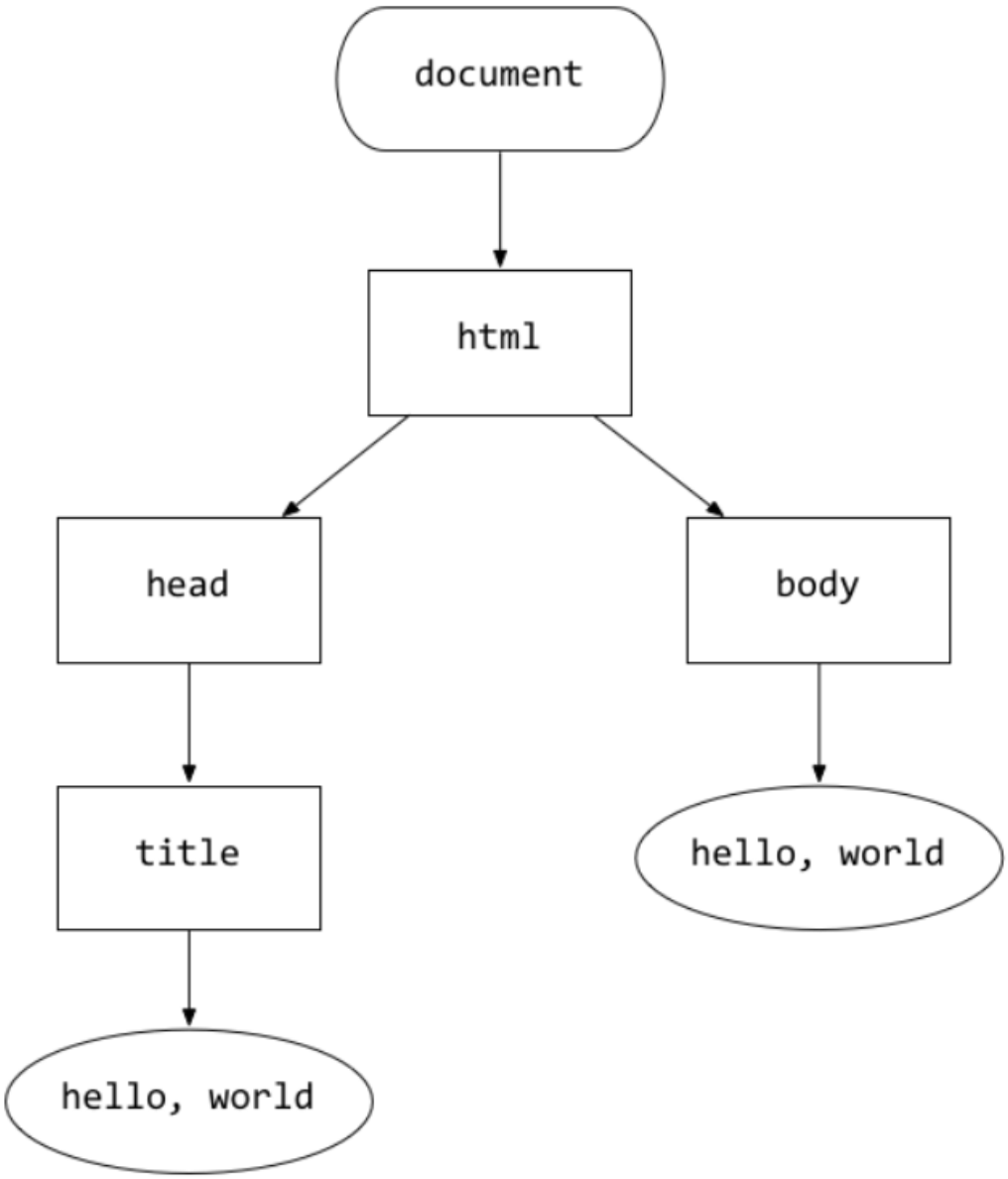
```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello,  world
    </body>
</html>
```

    # Notice the angle brackets and the words between those brackets, which we'll start calling **tags**. So `<head>` and `</head>` are the open and close tags, or the start and end tags, of an HTML **element** called head. The same applies to `<body>` and `<html>` and so forth.

- We'll look at more elements as we need them.
- Browsers interpret HTML pretty straightforwardly. When we say `<html>` that just means "here's the start of our HTML page", `<head>` means "this is the start of the head section", `</head>` means "this is it for the head section, stand by for something else", and so forth.
- Text that isn't in a tag, like `hello, world`, will just be displayed in the screen.
- Notice the indentation, where we indent every time a new tag is opened, and unindent when it is closed, just like curly braces.

    # With `<title>`, we used our judgment in keeping it on the same line since it looks cleaner, fits easily in one line, and only contains one inner element.

- The indentation could also help us think of the webpage as a tree:



The `html` tag is like the root of the tree.

We'll call this tree the **Document Object Model** (**DOM**) that represents the HTML.

`html` has two children, with `head` on the left and `body` on the right, and `head` has one child, `title`, which itself has a child, `hello, world`. The oval conveys that it's not a tag or element, but just text. These are all just arbitrary conventions that we use to represent an HTML document in a tree like this.

    # And as another note, `<!DOCTYPE html>` in our original source code is not a tag, but rather just a line placed there to indicate to the browser that this is an HTML5 file (different versions have different declarations[4]).

- Let's open `hello.html` with `gedit` in the appliance:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

    # This is a webpage that lives on the Desktop (or wherever you've saved it) and can be opened in Chrome with control-o:

    # Notice that the URL in the address bar reads `file:///home/jharvard/Desktop/hello.html`, indicating that we are looking at a file on our local hard drive, and no one else should be able to see it.

[4] http://en.wikipedia.org/wiki/Document_type_declaration

Servers and Permissions

- We can fix this by using a web server. The CS50 Appliance, apart from being able to compile and run C code in a standard environment, has also been configured to run standard, open-source server software, including Apache (the most popular web server software in the world) and MySQL (a database software we'll get to). Basically, we can use the appliance as a web server.
- First, we open our Terminal, and run the following:

```
jharvard@appliance (~): cd Desktop/
jharvard@appliance (~/Desktop): ls
hello.html
jharvard@appliance (~/Desktop): mv hello.html ../vhosts/localhost/public/
```

  # The last command moves our file `hello.html` into a folder in our home directory.

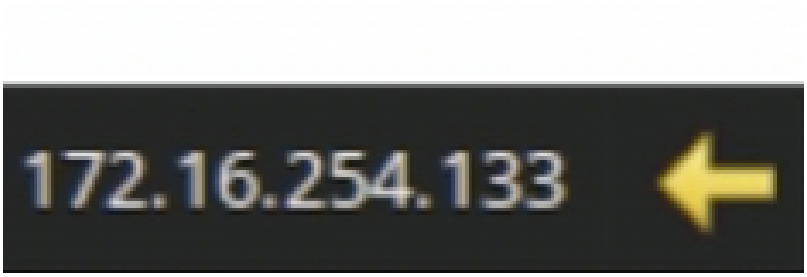
  # `vhosts` is the root folder that the appliance's server software looks into, `localhost` literally just means "this computer[5]," and `public` means that everything within that folder is public.

- We can check that it was indeed moved successfully with:

```
jharvard@appliance (~/Desktop): cd ../vhosts/localhost/public/
jharvard@appliance (~/vhosts/localhost/public/): ls
hello.html
```

- And now we can open a new tab in Chrome and enter `http://localhost/hello.html`, meaning we're visiting our own computer, and requesting the file `hello.html`.
- If we compare what we see now with to we saw earlier, the page looks the same, with the same "hello, world" text. But the difference now is that HTTP is being used.
- In the bottom right corner of the appliance, we have something that looks like this:

[5] http://en.wikipedia.org/wiki/Localhost

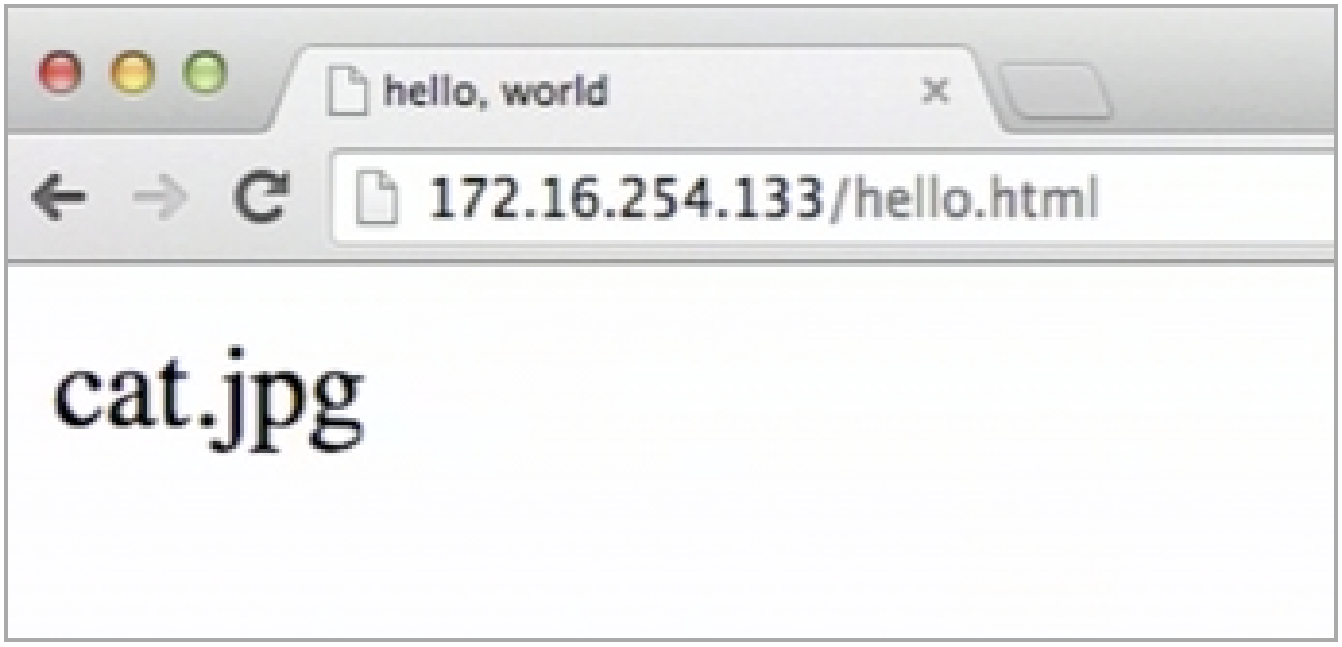


- This is a private IP address, so only the local network can access it. And it's the IP address of David's appliance, so yours might be different.
- We can open the version of Chrome on our Mac (that's running the appliance), go to `http://172.16.254.133/hello.html`, and see our webpage from our Mac. The web server doesn't have an easy-to-remember name, and isn't accessible by the rest of the Internet, but HTTP is indeed being used by Chrome on our Mac to communicate with the server running on our appliance.
- Let's go back to our appliance and `gedit`, and copy all the files from today's source code[6] to the `public` folder, including `cat.jpg`, and change `hello.html` to look like this:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        cat.jpg
    </body>
</html>
```

But when we save and reload, all we see is this:

[6] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/

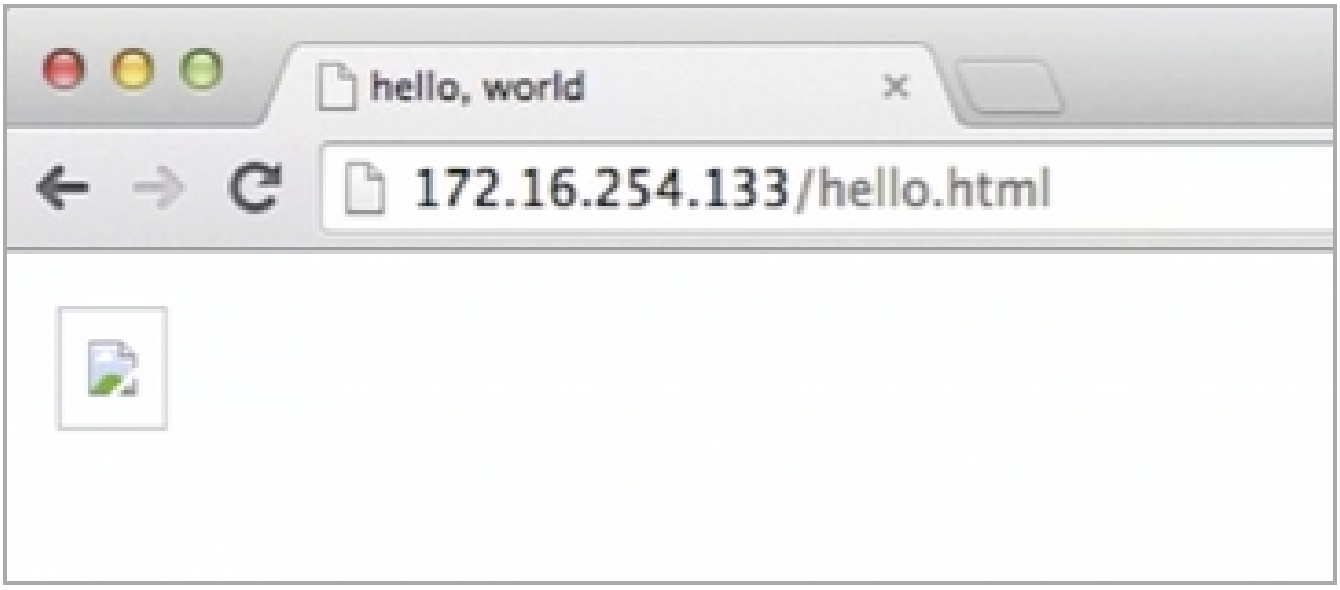


- We need another tag to tell the browser what we want to do:

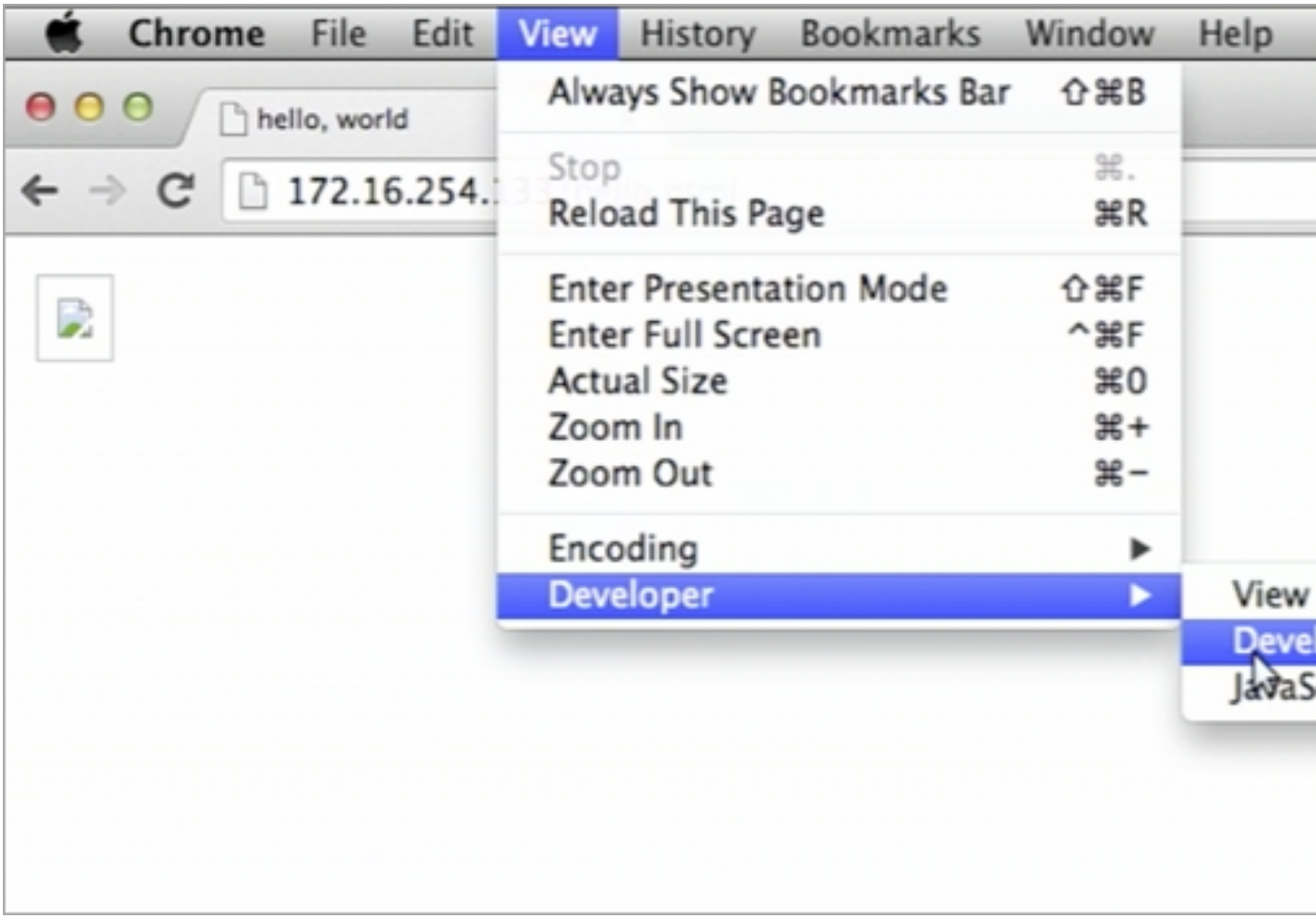
```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        <img src="cat.jpg"/>
    </body>
</html>
```

Note that, since the `img` tag is always going to be empty, it has the special syntax of being closed with a `/>` at the very end.

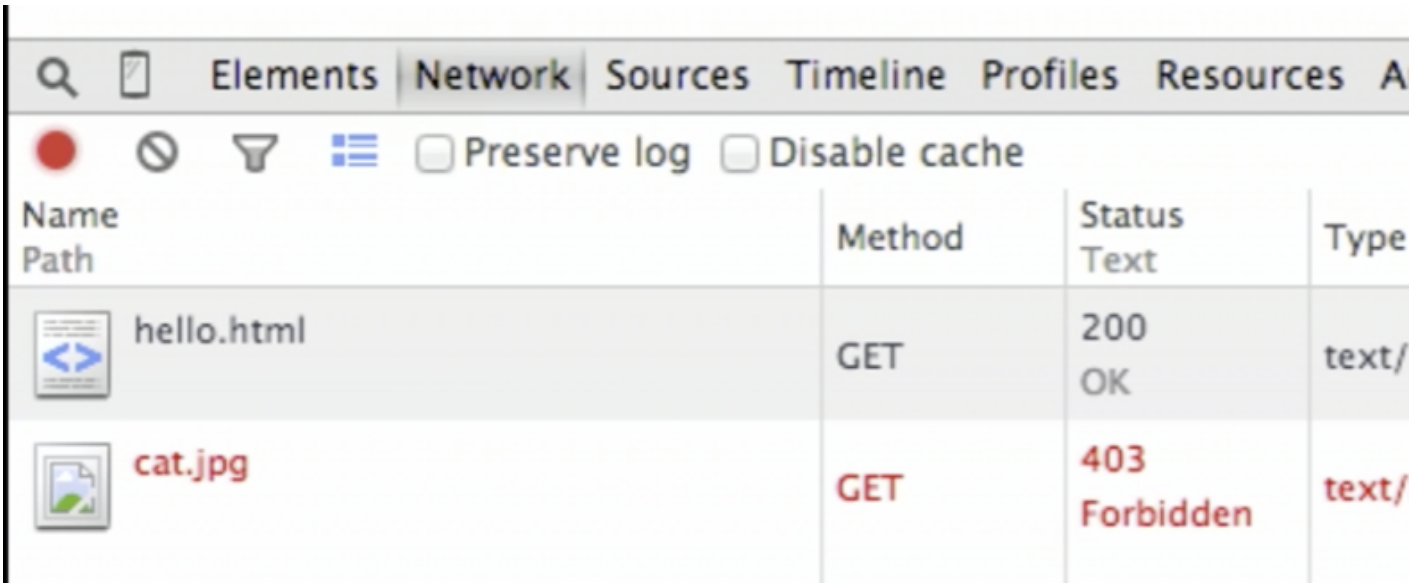
- But when we reload, we see something like this:



- We can go to **View**, **Developer**, **Developer Tools**:



- And if we go to the **Network** tab and reload the page, we can look at the requests being made:



- In particular, `hello.html` has a response of **200 OK**, but `cat.jpg` has **403 Forbidden**. (Recall that a **404** is returned when the file isn't found, so we know `cat.jpg` exists.)
- We can go back in our appliance, and run `ls -l` (bolded), which lists the files with more details:

```
jharvard@appliance (~/vhosts/localhost/public): ls
cat.jpg     css-1.html  css-2.html     hello.html  link.html  logo.gif
     search-0.html  search-2.html  search-3.html  table.html
css-0.html  css-2.css   headings.html  image.html  list.html
 paragraphs.html  search-1.html  search-3.css   search-4.html
jharvard@appliance (~/vhosts/localhost/public): ls -l
total 212
-rw------- 1 jharvard students 133986 Oct 22 13:26 cat.jpg
-rw------- 1 jharvard students    619 Oct 22 13:26 css-0.html
-rw------- 1 jharvard students    862 Oct 22 13:26 css-1.html
-rw------- 1 jharvard students    155 Oct 22 13:26 css-2.css
-rw------- 1 jharvard students    525 Oct 22 13:26 css-2.html
-rw------- 1 jharvard students    368 Oct 22 13:26 headings.html
-rw-r--r-- 1 jharvard students    143 Oct 22 13:27 hello.html
-rw------- 1 jharvard students    311 Oct 22 10:44 image.html
-rw------- 1 jharvard students    261 Oct 22 13:26 link.html
-rw------- 1 jharvard students    341 Oct 22 13:26 list.html
```

```
-rw------- 1 jharvard students  11988 Oct 22 13:26 logo.gif
-rw------- 1 jharvard students   1788 Oct 22 13:26 paragraphs.html
-rw------- 1 jharvard students    431 Oct 22 13:26 search-0.html
-rw------- 1 jharvard students    458 Oct 22 13:26 search-1.html
-rw------- 1 jharvard students    541 Oct 22 13:26 search-2.html
-rw------- 1 jharvard students     33 Oct 22 13:26 search-3.css
-rw------- 1 jharvard students    477 Oct 22 13:26 search-3.html
-rw------- 1 jharvard students   1126 Oct 22 13:26 search-4.html
-rw------- 1 jharvard students    714 Oct 22 13:26 table.html
```

We see `hello.html` and `cat.jpg`, but there are two `r`s for `hello.html` that `cat.jpg` doesn't (bolded).

- It turns out that we need to change the **mode**, or permissions, of the file, to allow the world to read (view) `cat.jpg`.
- For `hello.html`, the `r`s are what allow everyone to read it. So we can run `chmod`, which means "change mode," with `a+r`, which means "all+read," allowing everyone to read, `cat.jpg`:

```
chmod a+r cat.jpg
```

- Now if we run `ls -l` (bolded), we see that the permissions have been changed:

```
jharvard@appliance (~/vhosts/localhost/public): ls -l
total 212
-rw-r--r-- 1 jharvard students 133986 Oct 22 13:26 cat.jpg
-rw------- 1 jharvard students    619 Oct 22 13:26 css-0.html
-rw------- 1 jharvard students    862 Oct 22 13:26 css-1.html
-rw------- 1 jharvard students    155 Oct 22 13:26 css-2.css
-rw------- 1 jharvard students    525 Oct 22 13:26 css-2.html
-rw------- 1 jharvard students    368 Oct 22 13:26 headings.html
-rw-r--r-- 1 jharvard students    143 Oct 22 13:27 hello.html
-rw------- 1 jharvard students    311 Oct 22 10:44 image.html
-rw------- 1 jharvard students    261 Oct 22 13:26 link.html
-rw------- 1 jharvard students    341 Oct 22 13:26 list.html
-rw------- 1 jharvard students  11988 Oct 22 13:26 logo.gif
-rw------- 1 jharvard students   1788 Oct 22 13:26 paragraphs.html
-rw------- 1 jharvard students    431 Oct 22 13:26 search-0.html
-rw------- 1 jharvard students    458 Oct 22 13:26 search-1.html
-rw------- 1 jharvard students    541 Oct 22 13:26 search-2.html
-rw------- 1 jharvard students     33 Oct 22 13:26 search-3.css
-rw------- 1 jharvard students    477 Oct 22 13:26 search-3.html
```

```
-rw------- 1 jharvard students   1126 Oct 22 13:26 search-4.html
-rw------- 1 jharvard students    714 Oct 22 13:26 table.html
```

- We can also do the opposite with `chmod a-r cat.jpg` if we want to take that permission back.
- But let's not do that, and instead go back to our Chrome window, and if we reload the page, now we are able to see our grumpy cat.

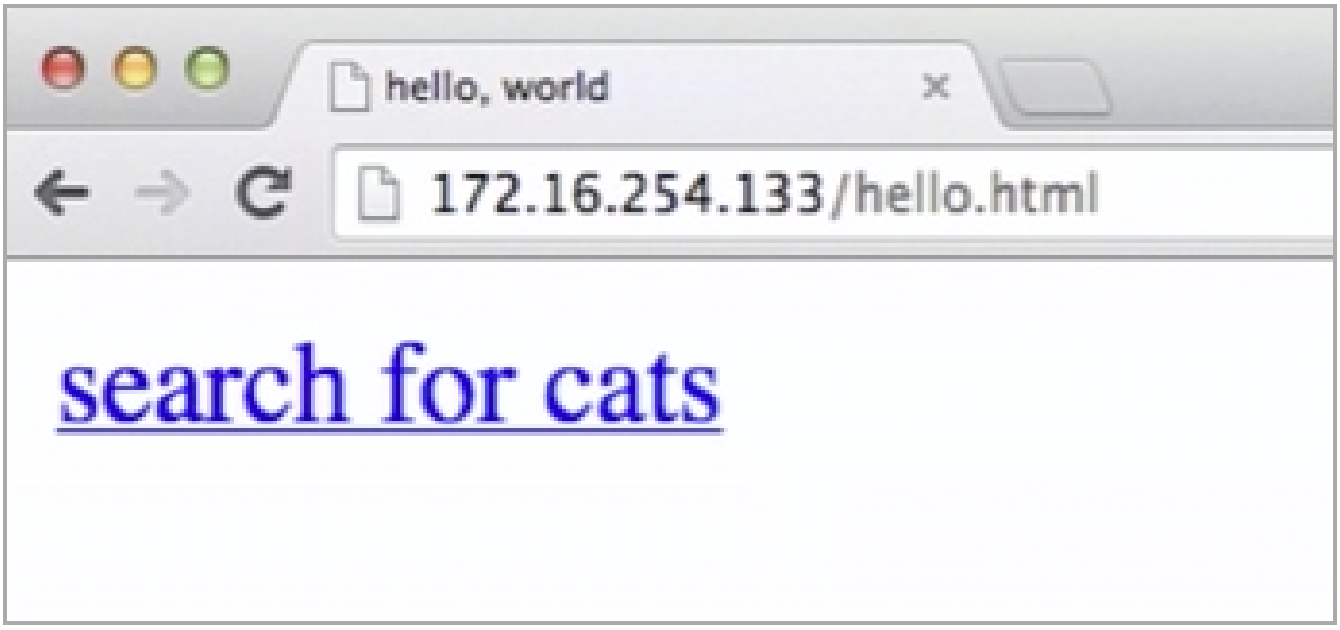
Links

- Let's try some other things. If we want a link, we could write something like this:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        <a href="http://www.google.com/">search for cats</a>
    </body>
</html>
```

    # `a` opens the tag and is short for "anchor", `href` stands for "hypertext reference", and `href` is not a tag but **attribute**, or something that can modify the behavior of a tag. In this case, `href` will tell the anchor tag the address that it should link to when it's clicked. The text in between, `search for cats`, is what will be shown to the human.

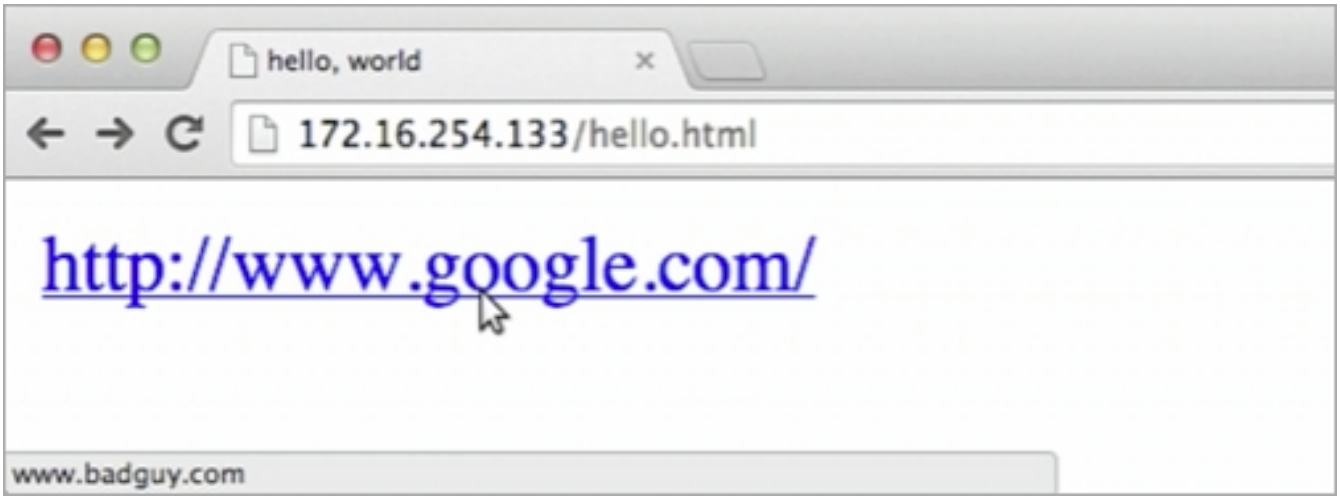
- We can see this if we save and reload:



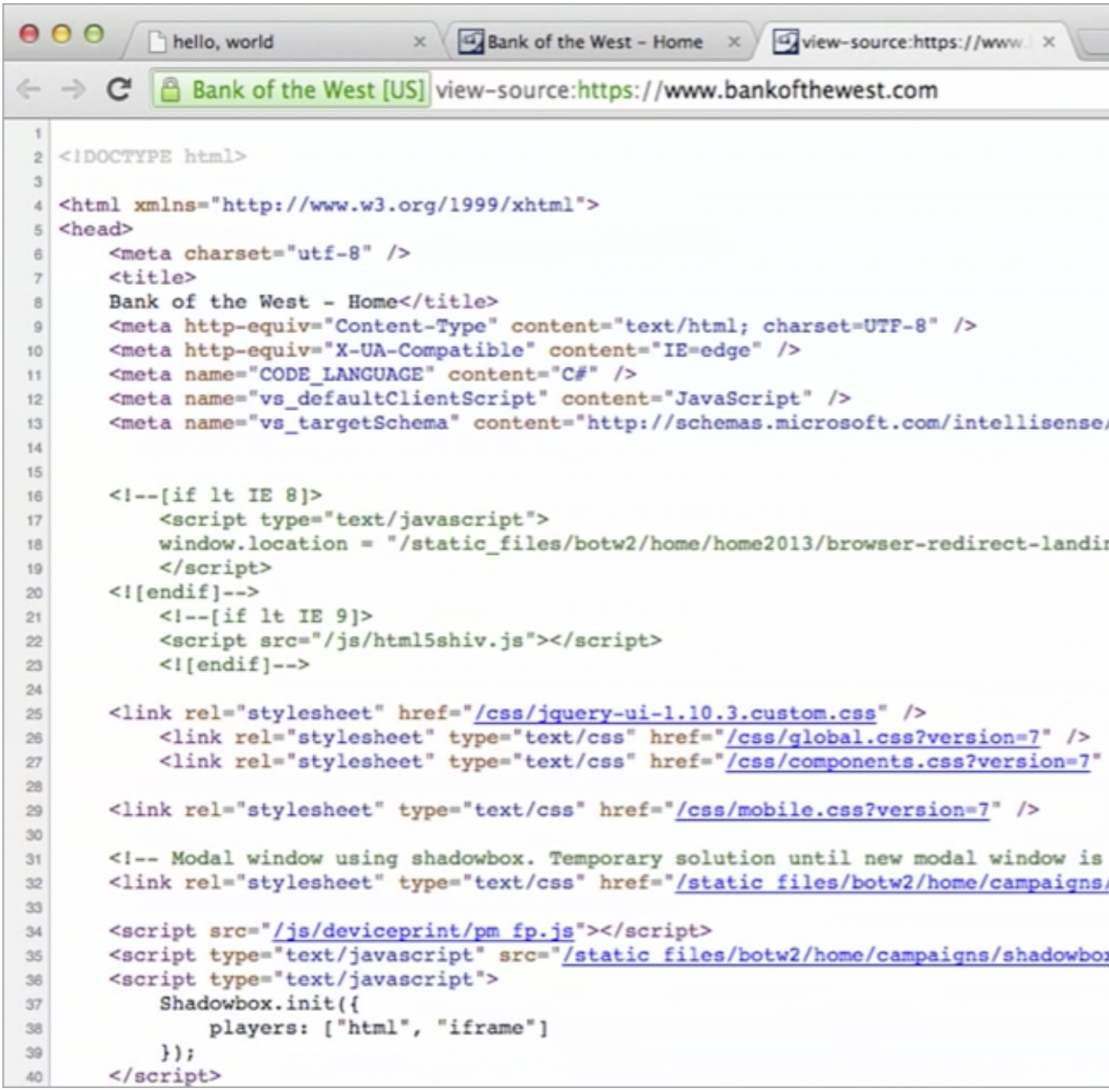
And notice that we can be tricky by making the link go to somewhere else:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        <a href="http://www.badguy.com/">http://google.com/</a>
    </body>
</html>
```

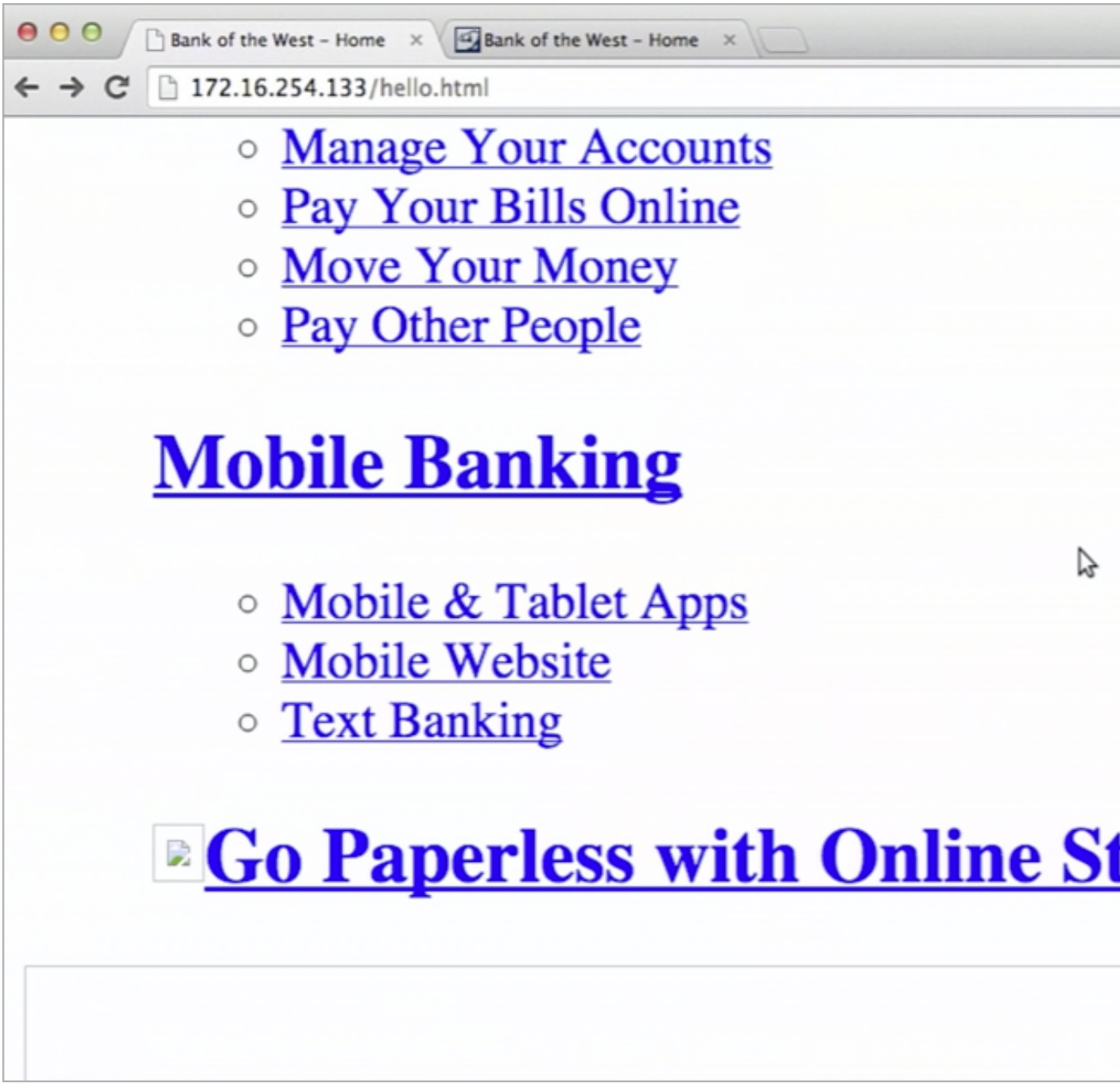
- Notice that now the link looks like it goes to `http://google.com/`, but only when we hover over it do we see that it doesn't:



- This is why you should never, ever click links in emails, because it's really easy to trick someone. (As an aside, someone actually bought the domain `http://www.bankofthevvest.com`, which looks very much like `http://www.bankofthewest.com` at first glance, in small font in an email, to try to trick people.
- In fact, if we were to go to `http://www.bankofthewest.com`, we can click **View**, **Developer**, **View Source**, and copy and paste the code we see into our own page:



- It’s not quite this easy, but we see that we are making progress toward making our own banking website:



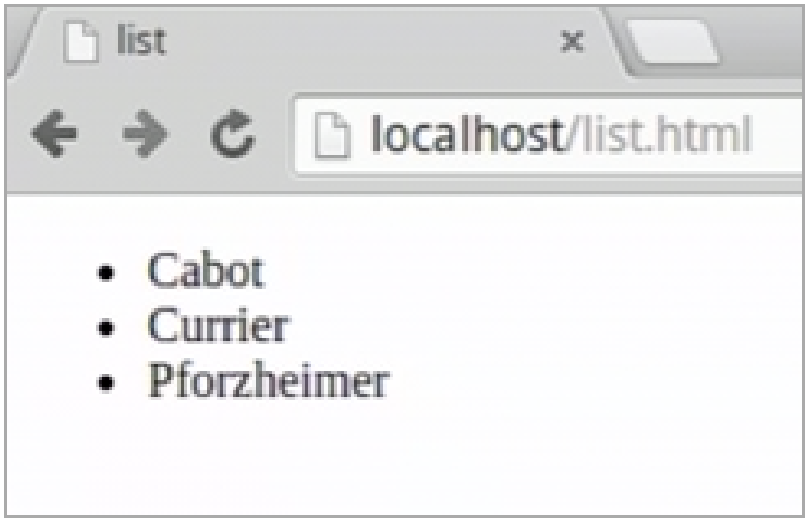
- The other files that the website might be using, include CSS files or images, probably need to be downloaded for the website to be complete.

Lists, Paragraphs, Tables

- Let's look now at `list.html`[7]:

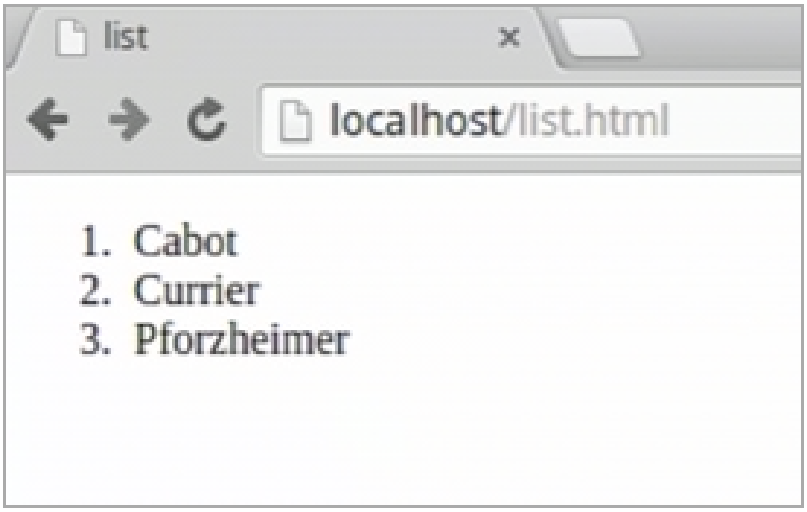
```
<!DOCTYPE html>

<html>
    <head>
        <title>list</title>
    </head>
    <body>
        <ul>
            <li>Cabot</li>
            <li>Currier</li>
            <li>Pforzheimer</li>
        </ul>
    </body>
</html>
```

The `<ul>` tag stands for **unordered list**, and `<li>` is a **list item**, making for a list that looks like:



- We can change the `<ul>` tag to an `<ol>` tag for an **ordered list**, which will now look like:

[7] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/list.html



- In `paragraphs.html`[8], we use the `<p>` tag to signify **paragraphs**:

[8] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/paragraphs.html

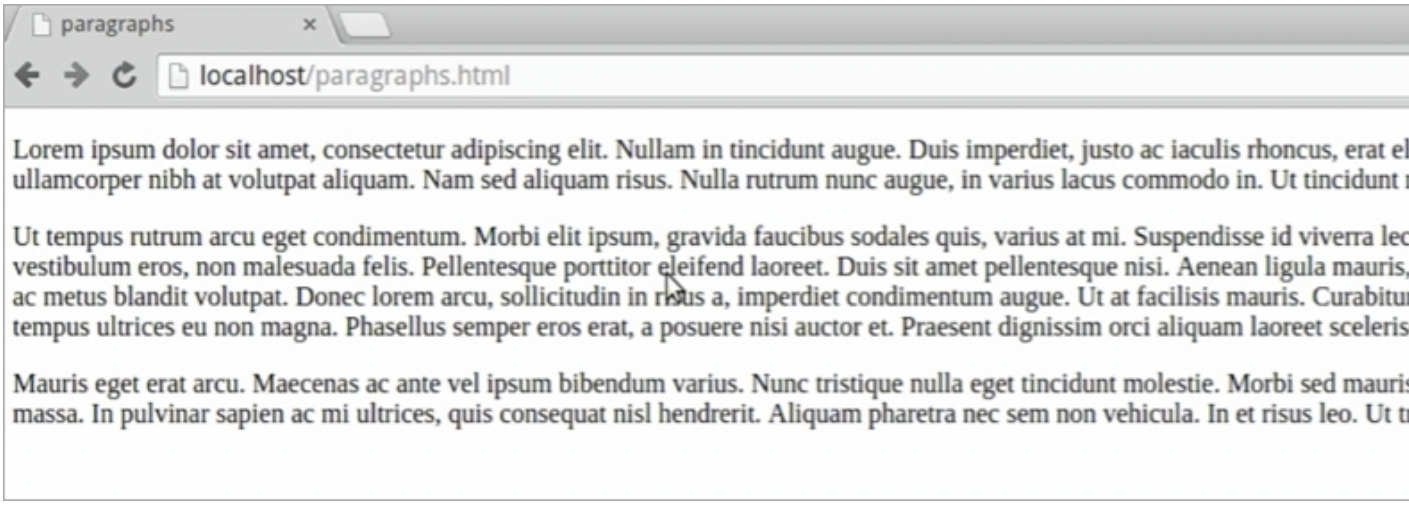
```
...
<p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam in
 tincidunt augue. Duis imperdiet, justo ac iaculis rhoncus, erat elit
 dignissim mi, eu interdum velit sapien nec risus. Praesent ullamcorper
 nibh at volutpat aliquam. Nam sed aliquam risus. Nulla rutrum nunc augue,
 in varius lacus commodo in. Ut tincidunt nisi a convallis consequat.
 Fusce sed pulvinar nulla.
</p>
<p>
    Ut tempus rutrum arcu eget condimentum. Morbi elit ipsum, gravida
 faucibus sodales quis, varius at mi. Suspendisse id viverra lectus.
 Etiam dignissim interdum felis quis faucibus. Integer et vestibulum eros,
 non malesuada felis. Pellentesque porttitor eleifend laoreet. Duis sit
 amet pellentesque nisi. Aenean ligula mauris, volutpat sed luctus in,
 consectetur id turpis. Phasellus mattis dui ac metus blandit volutpat.
 Donec lorem arcu, sollicitudin in risus a, imperdiet condimentum augue.
 Ut at facilisis mauris. Curabitur sagittis augue in dictum gravida.
 Integer sed sem sed justo tempus ultrices eu non magna. Phasellus semper
 eros erat, a posuere nisi auctor et. Praesent dignissim orci aliquam
 laoreet scelerisque.
</p>
<p>
    Mauris eget erat arcu. Maecenas ac ante vel ipsum bibendum varius.
 Nunc tristique nulla eget tincidunt molestie. Morbi sed mauris eu lectus
 vehicula iaculis ac id lacus. Etiam sit amet magna massa. In pulvinar
 sapien ac mi ultrices, quis consequat nisl hendrerit. Aliquam pharetra
 nec sem non vehicula. In et risus leo. Ut tristique ornare nisl et
 lacinia.
</p>
...
```

- We keep getting a **403 Forbidden**, so we'll run

```
jharvard@appliance (~/vhosts/localhost/public): chmod a+r *.html
```

  which will make all the `.html` files in our `public` folder readable.

- And now we see our paragraphs:

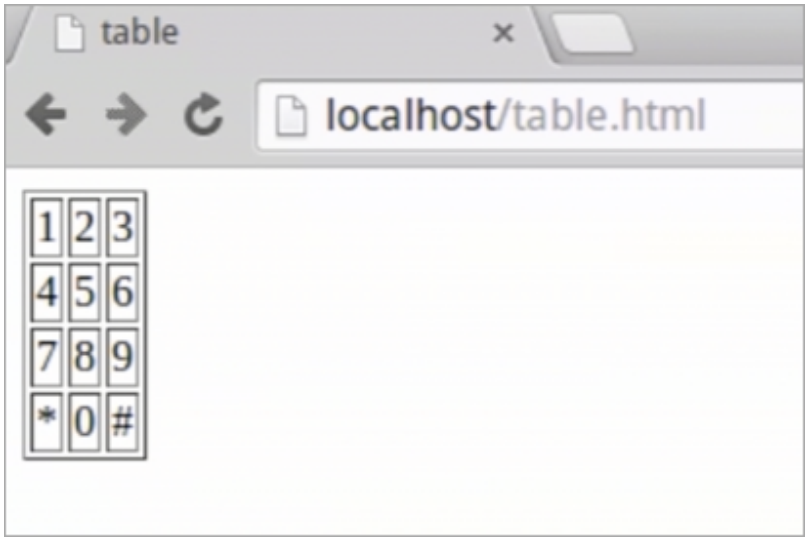


- We can also make **tables** with `table.html`[9]:

[9] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/table.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>table</title>
    </head>
    <body>
        <table border="1">
            <tr>
                <td>1</td>
                <td>2</td>
                <td>3</td>
            </tr>
            <tr>
                <td>4</td>
                <td>5</td>
                <td>6</td>
            </tr>
            <tr>
                <td>7</td>
                <td>8</td>
                <td>9</td>
            </tr>
            <tr>
                <td>*</td>
                <td>0</td>
                <td>#</td>
            </tr>
        </table>
    </body>
</html>
```

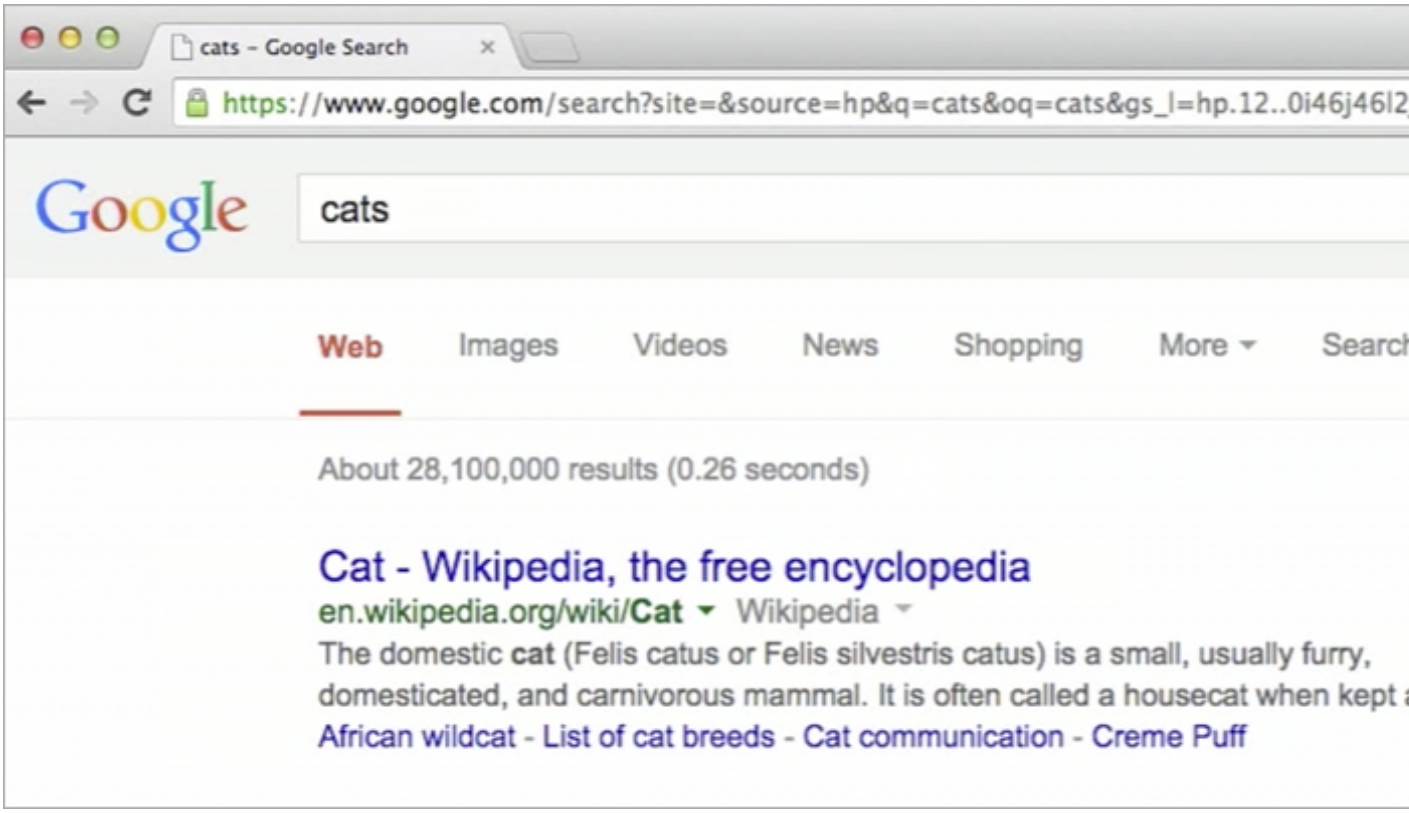
It's a bit more complicated, but `<tr>` stands for **table row**, and `<td>` for **table data**, or each cell in a table, and we see that it looks like this:



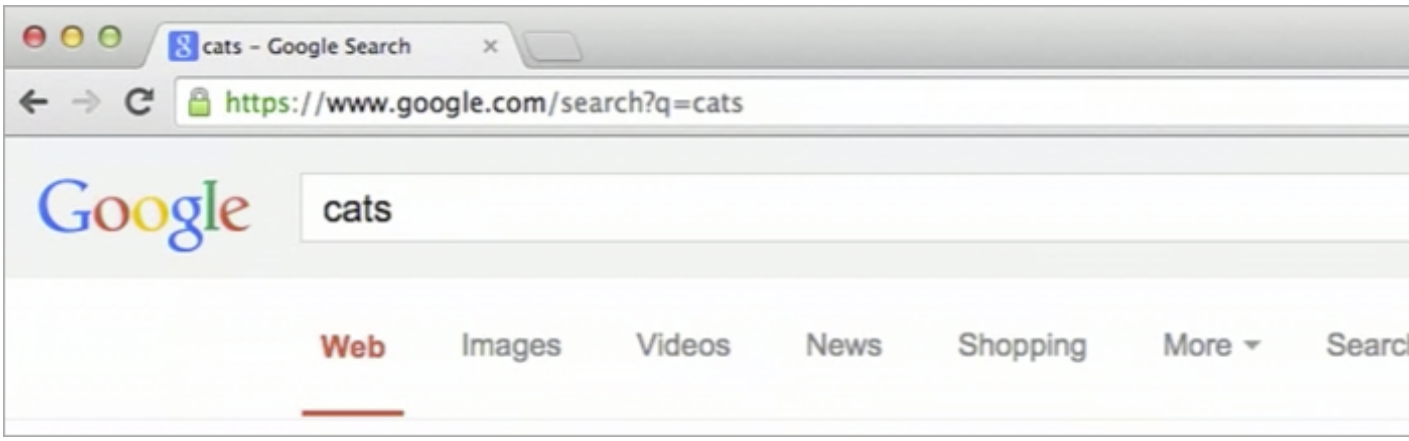
- HTML has lots of tags, and we can pick up the language as we need to, as long as we know the concepts of tags and attributes and how to form and structure HTML as we've seen.

Forms

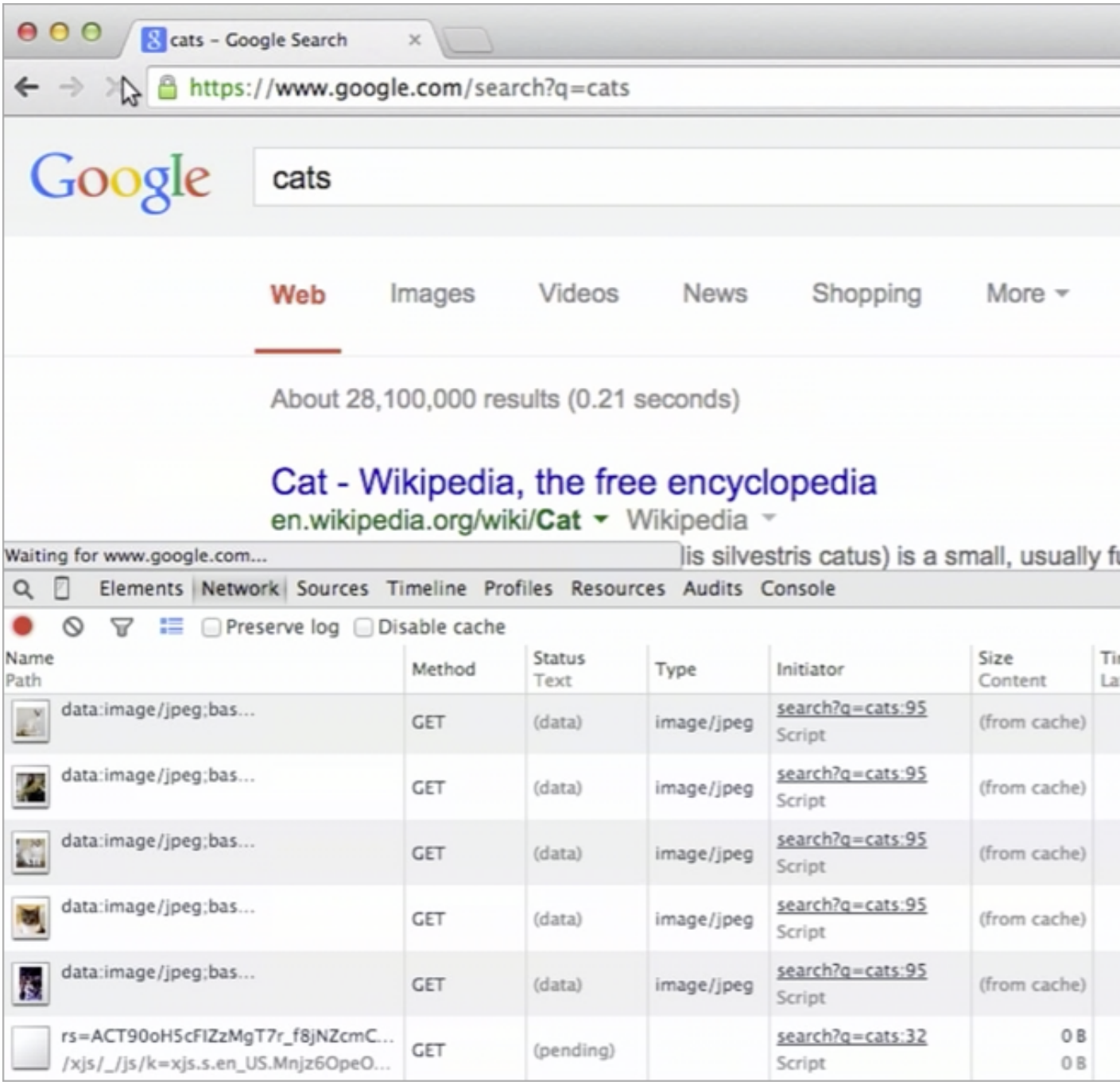
- Let's go back to our browser and search for cats. The URL becomes really long immediately:



- So let's delete what we don't understand, and see what happens:



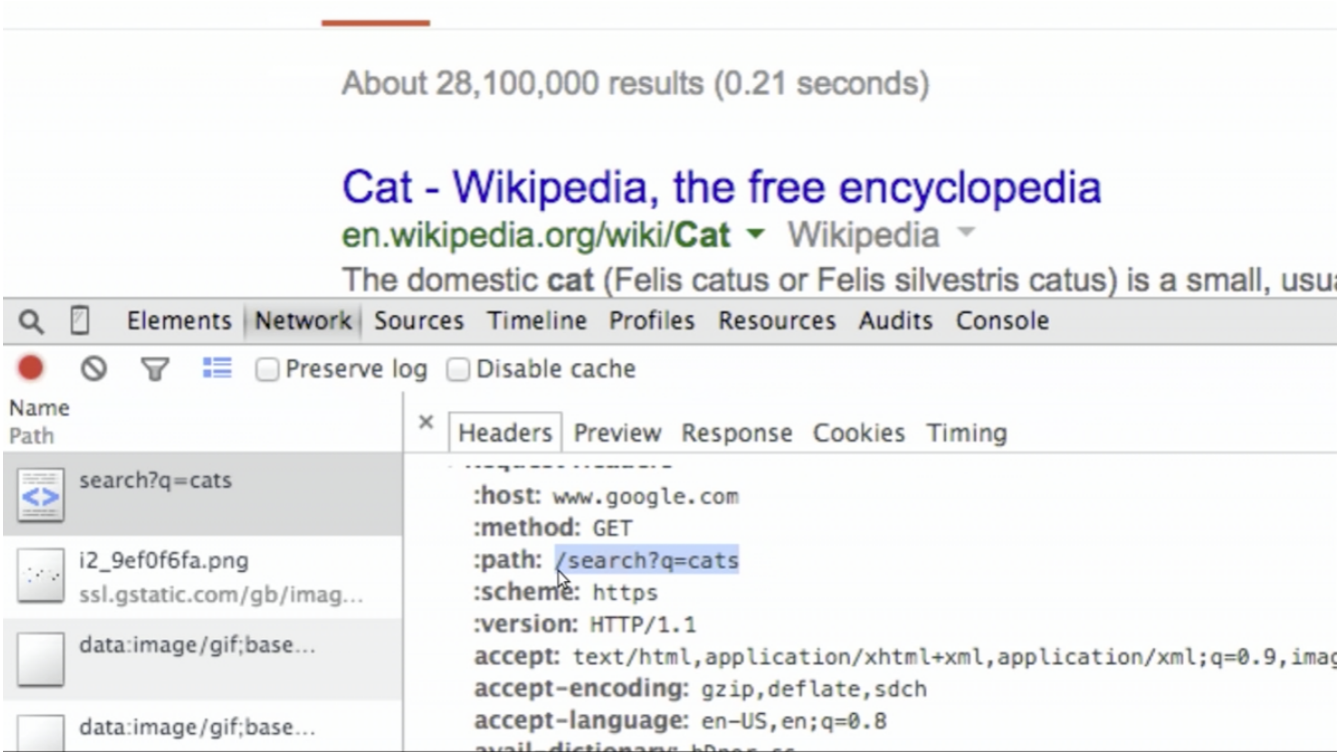
- It looks like the search still works, so Google is adding some unnecessary stuff. But now that we have a simple URL, we can open our friend **Developer Tools**, and look in the **Network** tab:



As an aside, we want to hold shift as we click the reload button, since it will make Chrome request the entire webpage again. Otherwise, browsers tend to **cache**, or save, information, to make loading faster, but we want to start over here to see what's happening.

We see all the requests, that are not only for text, but also images, icons, and pieces (like Scratch pieces), that the browser puts together to make a complete page.

- When the browser gets that initial HTML file, it goes through and looks for image tags and other tags that requires other files, and uses HTTP (think of the envelopes) to request those pieces and place them where they should go.
- We can also see that we've sent a request with `HTTP/1.1` and the path `/search?q=cats`



- So it seems like we can create our own search engine front-end (user interface) knowing that the requests are in this format.
- Let's look at `search-0.html`[10]:

[10] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/search-0.html

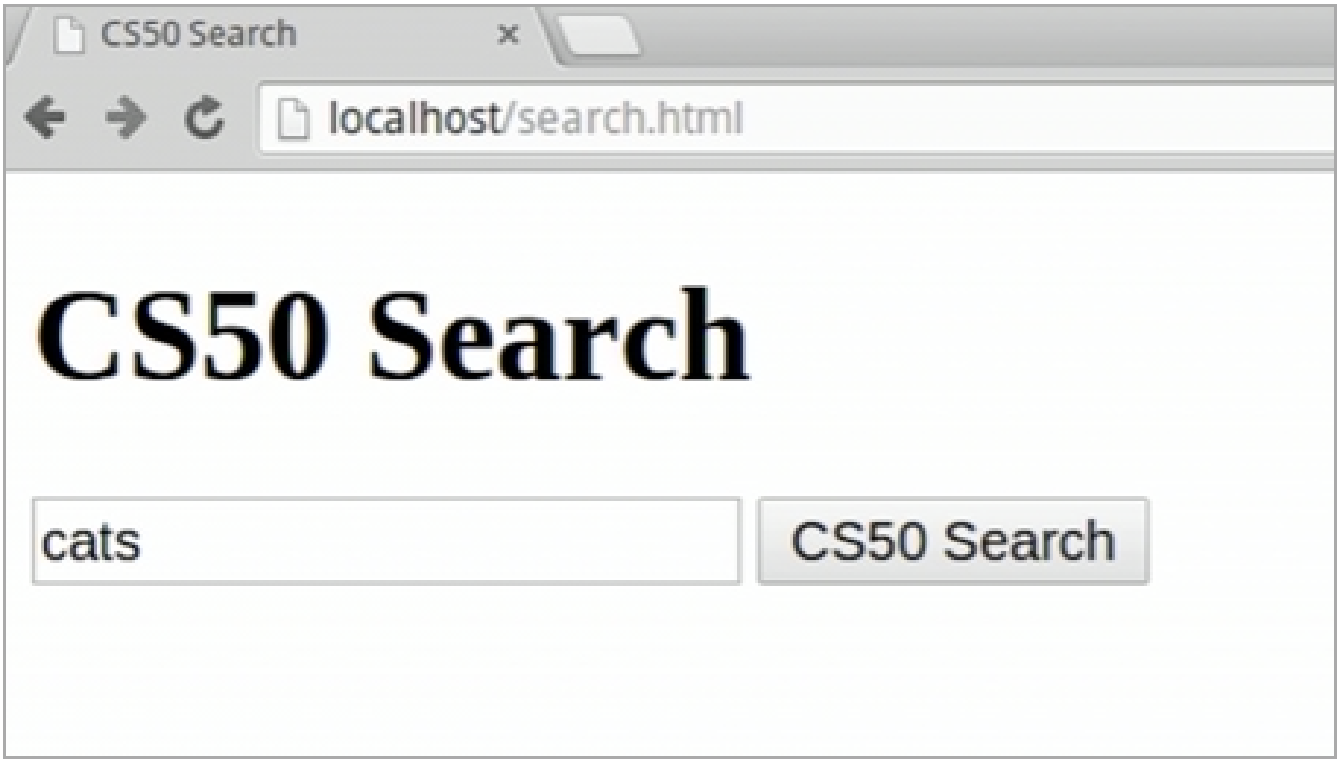
```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1> ❶
        <form action="https://www.google.com/search" method="get"> ❷
            <input name="q" type="text"/> ❸
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

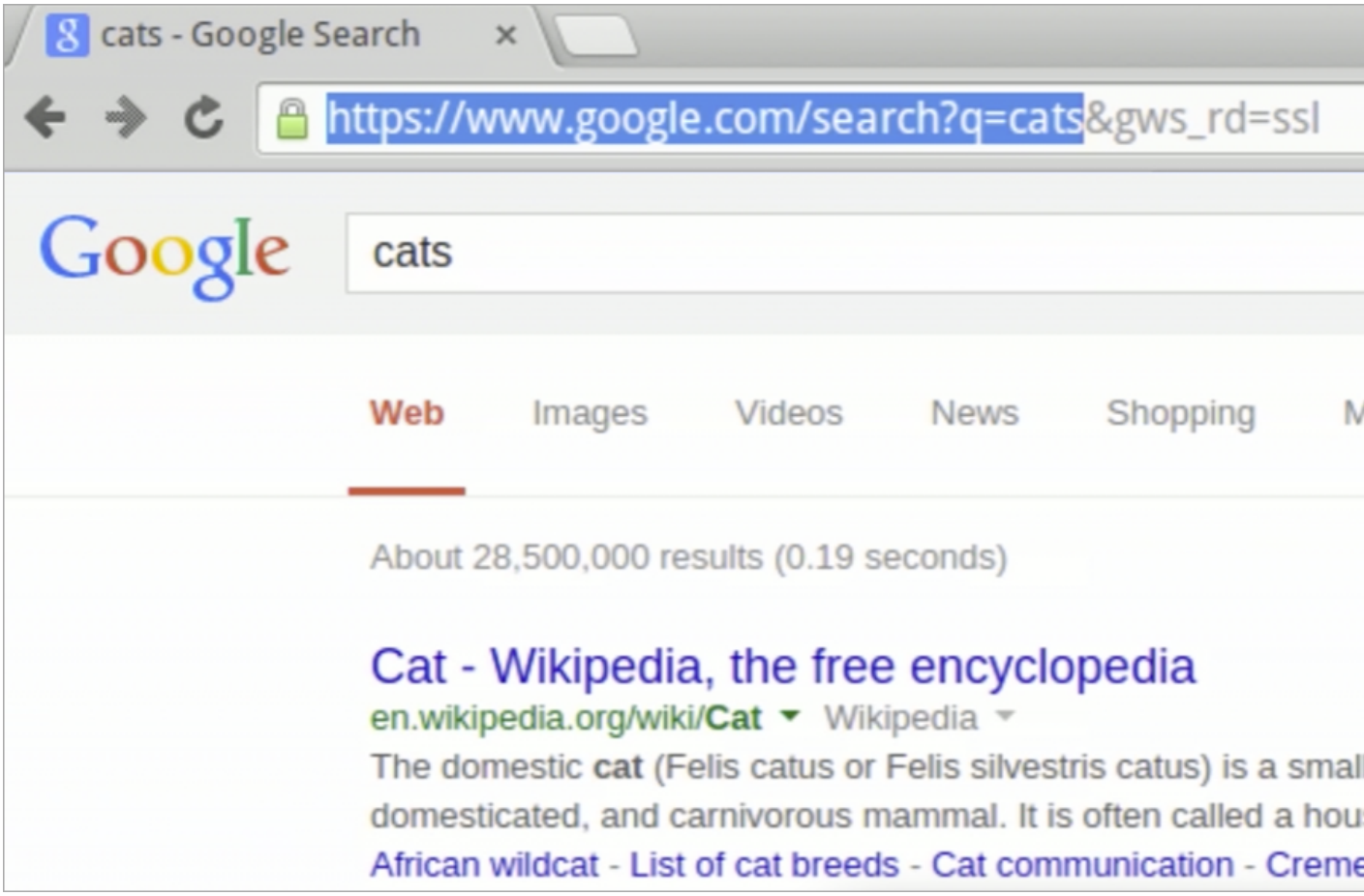
In line 6, we use the `h1` tag, which specifies that it is a **heading** that formats the text to be bigger and bolder.

In line 7, we open the `form` tag, with an `action` attribute that we set to `https://www.google.com/search` based on what we've seen in their URLs. (`method="get"` refers to using the GET method in the HTTP protocol, as opposed to something like POST, which we'll see later.)

In line 8, we create an `input` that accepts `text`, and it should be called `q` based again on what we saw in the original URL, `https://www.google.com/search?q=cats`. (`q` is probably just short for query.)

- Now if we go to `search-0.html` in our browser and input `cats`, we are indeed brought to the page we expected:


CS50 Search
localhost/search.html
CS50 Search
cats
CS50 Search



Google automatically adds the extra parameter because we're visiting the secure site, but notice that we've built this URL.

- In general, the `form` allows us to choose a URL to go to when it's submitted, as well as appending a parameter based on its input.
- But we can make it look a bit better. Let's open `search-1.html`[11]:

[11] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/search-1.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50 Search</title>
    </head>
    <body style="text-align: center"> ❶
        <h1>CS50 Search</h1>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text"/>
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

We see that line 5 is a bit strange, and what it does is pretty straightforward, centering everything on the page:

CSS

- It turns out that we're now using another language to do this: **Cascading Style Sheets** (**CSS**).

- CSS styles the entire web today, and much like HTML it is a simple language, with properties that can be learned as needed. The basic structure of CSS uses key-value pairs, where "key" is some property, and "value" is just the value of that property.
- For example, in `search-1.html`, the key is `text-align` with the value `center` that we combine with the `:` syntax. We can look up that the value for `text-align`, for instance, could be `left`, `right`, or `center`:

```
...
    <body style="text-align: center">
...
```

- Let's open `search-2.html`[12]:

```
<!DOCTYPE html>

<html>
    <head>
        <style>

            body
            {
                text-align: center;
            }

        </style>
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text"/>
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

We've done a little more here, by placing a `<style>` tag in the `<head>` section of the page.

[12] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/search-2.html

  # Within the `<style>` tag, we've put the name of the tag we want to change, `body`, above some curly braces, with the property and value within.

  # The end result is identical, but this is better design as the CSS is factored out, or separated from, the HTML, and allows us to reuse and change code more easily. If we wanted to center multiple elements, we wouldn't have to type `style="text-align: center"` for all of them, but just put it in one place.

- The best design, though, is with `search-3.html`[13]:

```
<!DOCTYPE html>

<html>
    <head>
        <link href="search-3.css" rel="stylesheet"/> ❶
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text"/>
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

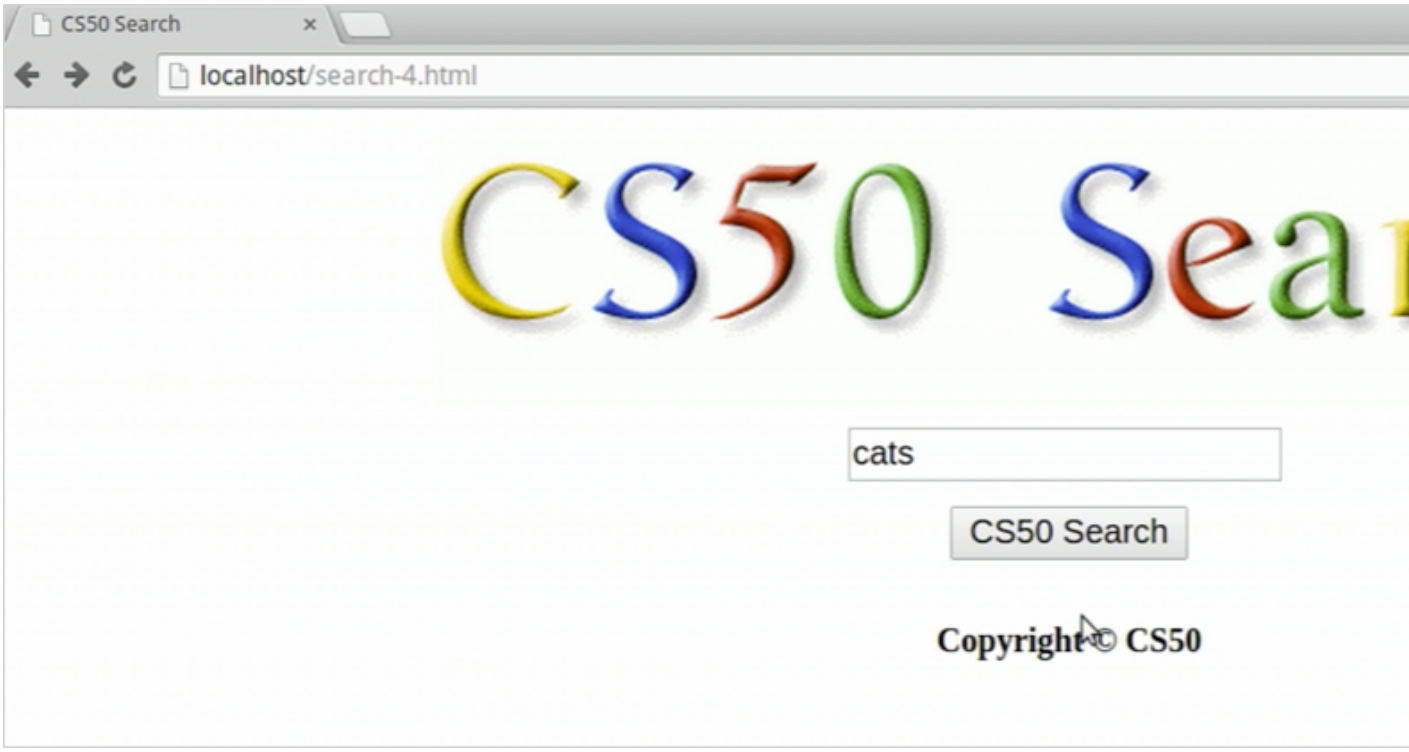
  # The page is cleaned up a bit here, with line 3 containing a `<link>` tag that links to another file, `search-3.css`, that looks like this (`rel` indicates the relationship between that file and the current one):

```
body
{
    text-align: center;
}
```

  # The end result will be again the same (as long as we've set the correct permissions with `chmod`)!

[13] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/search-3.html

- Let's look now at `search-4.html`[14] in a browser:



Now we're getting pretty close to a 1999 (or even 2014) version of Google!

- Let's glance through the source code (with CSS within the same file for learning convenience for now):

[14] http://cdn.cs50.net/2014/fall/lectures/7/w/src7w/search-4.html

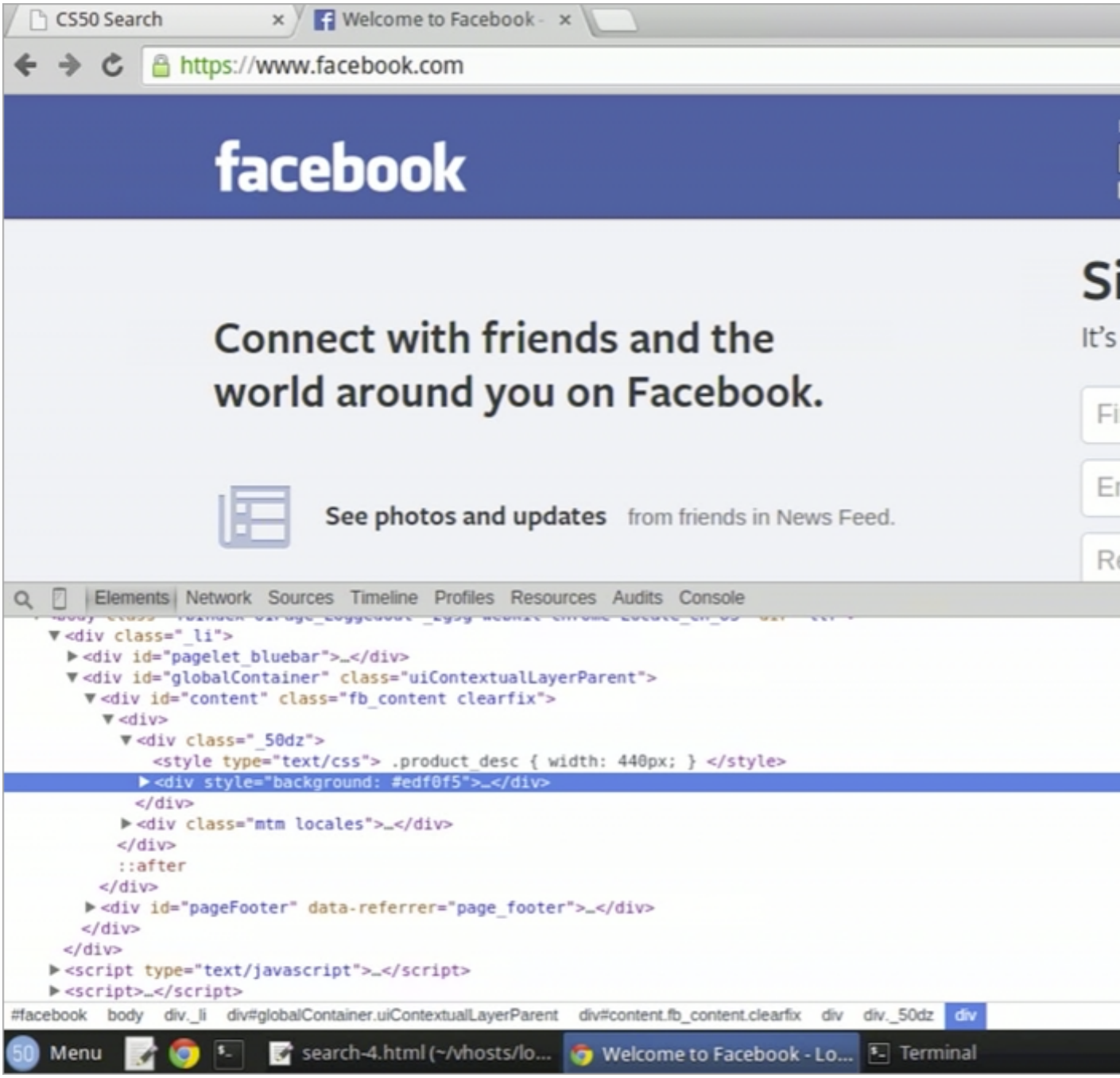
```
<!DOCTYPE html>

<html>
    <head>
        <style>

            #header
            {
                text-align: center;
            }

            #content
            {
                text-align: center;
            }

            #content input
            {
                margin: 5px;
            }

            #footer ❶
            {
                font-size: smaller;
                font-weight: bold;
                margin: 20px;
                text-align: center;
            }

        </style>
        <title>CS50 Search</title>
    </head>
    <body>
        <div id="header"> ❷
            <img alt="CS50 Search" src="logo.gif"/>
        </div>
        <div id="content"> ❸
            <form action="https://www.google.com/search" method="get">
                <input name="q" type="text"/>
                <br/>
                <input type="submit" value="CS50 Search"/>
            </form>
        </div>
        <div id="footer"> ❹
            Copyright &#169; CS50
        </div>
    </body>
```

    # We see some new tags, like `<div>` in line 42, that just means a new division, or region of the page. The `id="footer"` gives it an identifier that is `footer` in this case, which we can use elsewhere (like in line 20) to refer to it.

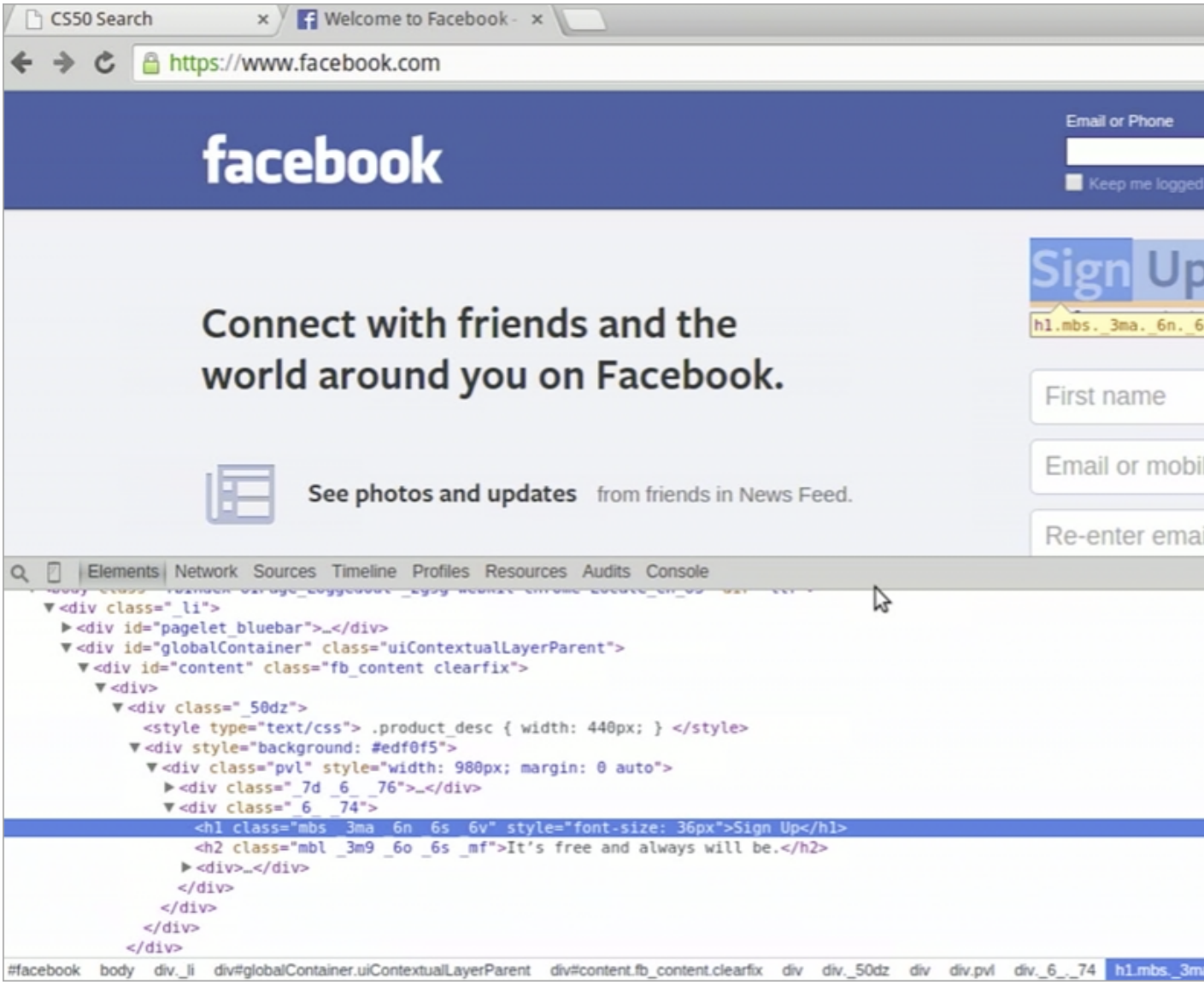
    # We also see `<div id="header">` and `<div id="content">` on lines 32 and 35 that divides the page invisibly into three sections, which allow us to change their styles independently.

    # Notice that we can set lots of settings for `footer` that isn't applied to `header` or `content`, making CSS quite powerful.

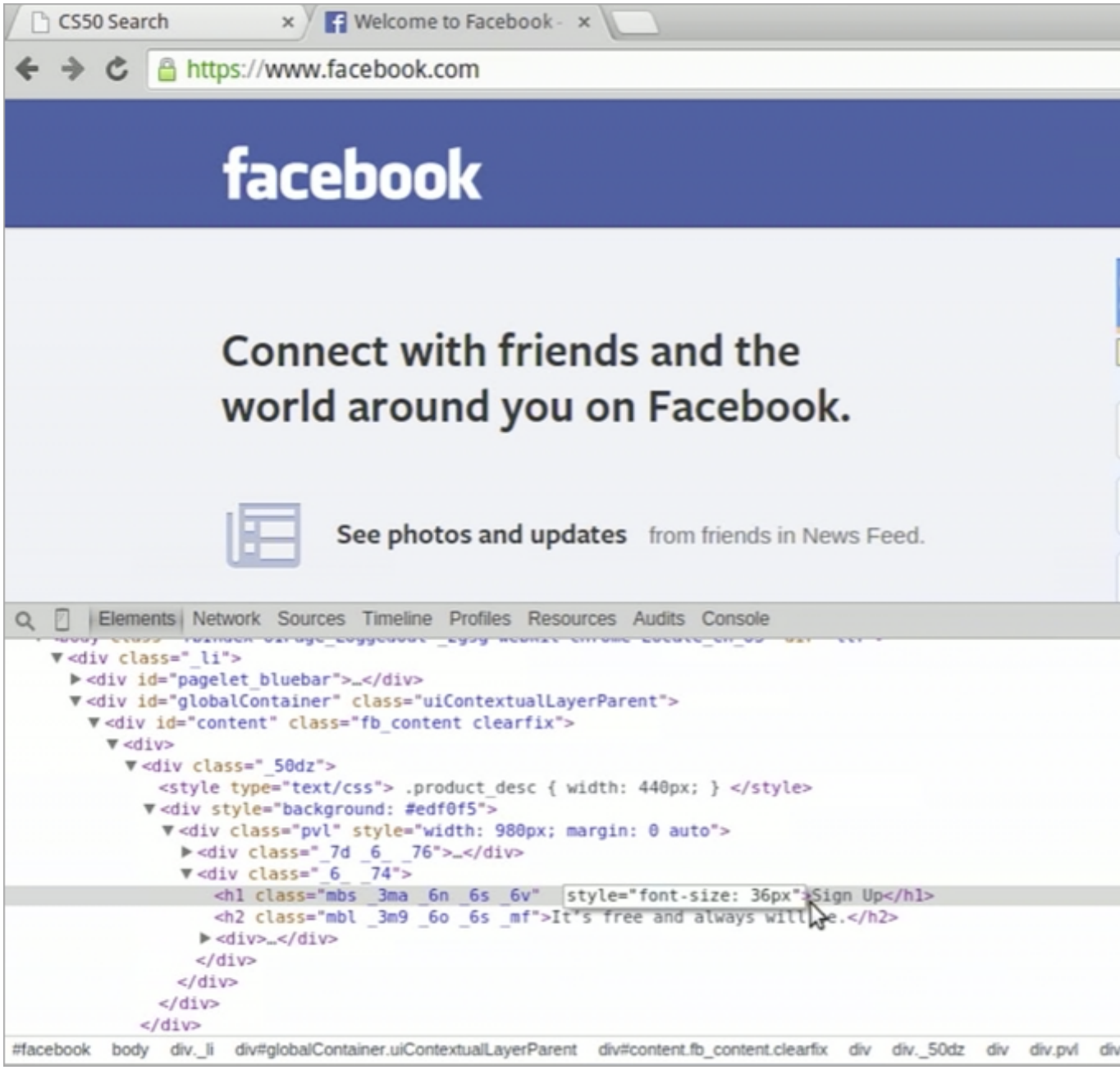
- Let's go to facebook.com and open our **Developer Tools**:



- We can right-click a word like `Sign` and click **Inspect Element** that bring us directly to the `<h1>` tag in the HTML for that word:



We can actually double-click that element and change the value (at least for us, locally).



And the sidebar on the right allows us to do the same:

- This makes it easier for us to debug and try things on our own webpages, if we want to create our own.
- We'll also want to run websites that are completely our own, where the form no longer asks Google but our own site, with something like this:

```
...
<form action="search.php" method="get">
    <input name="q" type="text"/>
    <br/>
    <input type="submit" value="CS50 Search"/>
</form>
...
```

PHP

- We'll need to introduce another language, **PHP**, that you may notice is the ending of the page `search.php`. HTML and CSS are just aesthetic languages that allow us to structure and style a page, but not programming languages with complicated logic.
- PPHP is a scripting language that is lighter-weight than C. It's an **interpreted language**, which means it's not compiled but executed line by line by an interpreter, a program that reads each line and does what it says (like Python and Ruby and Perl and others).
- There are many similaries to C, but some differences include that there's no `main` function anymore, and variables look like this:

```
$s = "hello, world";
```

  # Notice that there's **weak typing**, which means variables have types, but we no longer need to specify them (a blessing and a curse!).

  # The dollar sign is just PHP syntax for the start of a variable name.

- Apart from that, PHP is quite similar. This is a condition in PHP:

```
if (condition)
{
    // do this
}
else if (condition)
{
    // do that
}
else
{
    // do this other thing
}
```

- Boolean expressions:

```
if (condition || condition)
{
    // do this
}
```

```
if (condition && condition)
{
    // do this
}
```

- Switches:

```
switch (expression)
{
    case i:
        // do this
        break;
    case j:
        // do that
        break;
    default:
        // do this other thing
        break;
}
```

- Loops:

```
for (initializations; condition; updates)
{
    // do this again and again
}
```

```
while (condition)
{
    // do this again and again
}
```

```
do
{
    // do this again and again
}
while (condition);
```

```
foreach ($numbers as $number)
{
    // do this with $number
}
```

That last one is a bit different. It's a convenient feature whereby, if `$numbers` is an array, then `$number` will give you each element of the array one by one, like `$numbers[$i]`, only we didn't need to set up the variable `$i`.

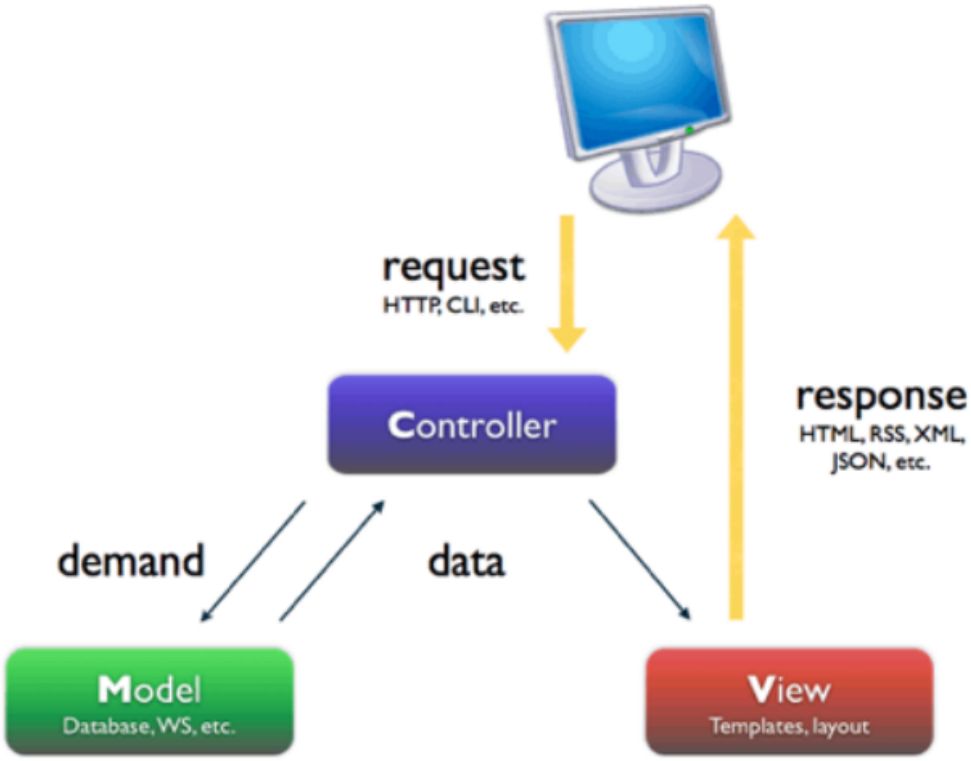
- We also tend to pre-initialize arrays in square brackets:

```
$numbers = [4, 8, 5, 16, 23, 42];
```

- And we also have **associative arrays** in PHP, which is like a hash table:

```
$quote = ["symbol" => "FB", "price" => "79.53"];
```

- We'll look at that more closely on Monday, but the takeaway is that lots of languages will start to abstract away these structures, meaning someone else has created those features for you, and you can use them in your own programs without having to implement them all over again.
- We could redo all of our problem sets this semester in PHP, perhaps a bit more easily, but what we eventually want to do is to write web applications.
- We'll introduce a new concept, MVC (model-view-controller) that factors code in a way that many websites tend to use:



- This might seem overwhelming, but realize that over 90% of all students who take CS50 end up making final projects based on web programming.