Web Development

This is CS50. Harvard University. Fall 2015.

Cheng Gong

Table of Contents

Databases, Continued

- Imagine that you're living with a roommate, and we have a shared fridge. You come home one day to see that there's no more milk, so you leave to go buy some more. But while you're at the store, your roommate comes back, and also leaves to some other store to buy more milk, and now you have twice as much milk as you needed.
- The problem here is that we didn't leave a note, or communicate otherwise with our roommate. This relates to databases where we might have lots of users all talking to the same central source. For example, imagine someone trying to withdraw money from a joint account at an ATM while someone else on that same account is trying to do the same on another ATM. And if they do it at the same time, the central computer keeping track might only subtract the amount once, or allow both ATMs to release money even though the total withdrawn might be higher than what's in the account.
- One solution might be locking the account so only one request per account can be processed at a time. We need to be careful in making sure that we have **atomicity**, or that any operations we do to a database as part of a request happen together. For example, if we want to check the balance of an account, subtract some amount, and save the new balance, we would want that to happen all at once in the same **transaction**.
- If we aren't careful with this, we might have a **race condition** where the timing of multiple requests might lead to different outcomes. For example, we wouldn't want to allow two people to register for the same username on a website if they tried to do so at the same time, and we might solve that issue by locking the database or row so each transaction can complete.

- Another issue might be accidentally allowing **SQL injection** in our code, where users (or bad guys) might type in SQL code like `INSERT` or `DELETE` into web forms like username and password, but the server accidentally executes that code.
- For example, we might have a SQL query that looks like:

```
SELECT * FROM users WHERE username = '$username' AND password =
 '$password'
```

  # `$username` and `$password` are just variables that stored whatever someone typed in the form.
  # If we type in `whatever' OR '1' = '1` for our password, we'll end up with an expression in our code that actually looks like `SELECT * FROM users WHERE username = '$username' AND password = 'whatever' OR '1' = '1'`. And that would select all the users with the username we put in, even though the password didn't match because we also conditioned on `'1' = '1'`.
  # And there could be even more queries added in, because we're trusting what the user has inputted.

- So to try to prevent this, we would want to **escape** characters, or prevent the character from being a part of code. For example, in our query, if we added `\'` to our single quotes, we would have made the single quote just a single quote, rather than ending the string.
- We look at this XKCD comic[1] as another example.
- And to be clear, the backslash characters don't come from users, but rather our own code before we attempt to run the SQL query.
- So we learned about SQL, and another trend these days is NoSQL, where data is no longer stored in rows and columns but just objects that might look like:

```
{
  "_id": "02134",
  "city": "Allston",
  "loc": [
    -71.132866,
    42.353519
```

[1] https://xkcd.com/327/

```
  ],
  "pop": 23775,
  "state": "MA"
}
```

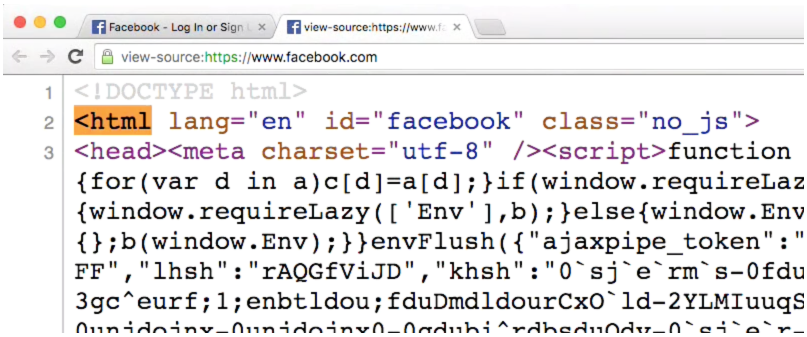
And this allows for our data to be hierarchical. Notice that inside we have a bunch of **key-value pairs** where a quoted word on the left like `city` has some value associated with it.

The value for `loc` is an array, in this case an array of size 2.

- And we can also store objects with different types of data inside.

Web Development

- **HTML**, Hypertext Markup Language, is used to write webpages. If you recall our metaphor with the envelopes, HTTP is the protocol for sending messages, but the content of those messages are written in HTML. For example, in Chrome, we can go to `View`, `Developer`, `View Source` when we're at some webpage, to see its source:



We notice some common formats like using `<` and `>` to mark the start and end of **tags**.

- The simplest webpage, meanwhile, might look like this:

```
<!DOCTYPE html>

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

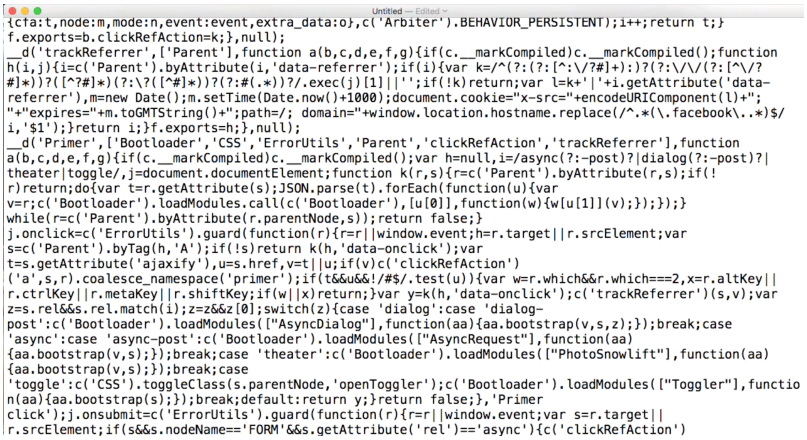
```
        hello, world
    </body>
</html>
```

So the first line is just declaring that this is an HTML file, and we start the page with `<html>`. Then we have `<head>` for the head of our webpage with a `<title>` inside. The text `hello, world` not inside a tag will just be text as part of our page, in this case in the title of the page. And notice that we close tags with a `/`.

Then we have a `<body>` tag that we put some text in, and then close.

Notice that we have symmetry in how we close tags.

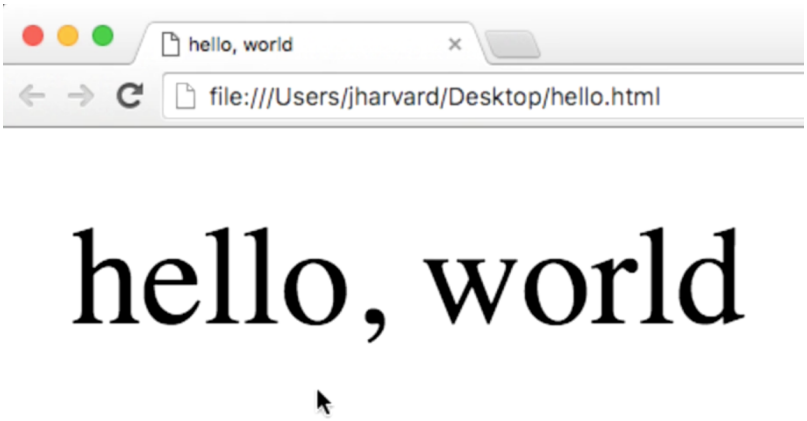
- And as an aside, Facebook's source doesn't have these new lines and indentation because their code probably removes the whitespace to decrease the size of the page and make sending it faster, since our browsers don't take that into account when rendering a webpage.
- And even though we can see Facebook's generated HTML code for their pages, we can't see their back-end code in how it was generated, etc. There's also CSS code, for styling the webpage, and JavaScript code, which is more important since it actually has programmed logic that implements functionality of the webpage.
- To keep that code short and difficult to reverse-engineer, after the code is written it is **obfuscated** by some program that just shortens the code:



```
{cfa:t,node:m,mode:n,event:event,extra_data:o},c('Arbiter').BEHAVIOR_PERSISTENT);i++;return t;}
f.exports=b.clickRefAction=k;},null);
__d('trackReferrer',['Parent'],function a(b,c,d,e,f,g){if(c.__markCompiled)c.__markCompiled();function
h(i,j){i=c('Parent').byAttribute(i,'data-referrer');if(i){var k=/^(?:(?:[^:\/?#]+):)?(?:\/\/(?:[^\/?
#]*))?([^?#]*)(?:\?([^#]*))?(?:#(.*))?/.exec(j)[1]||'';if(!k)return;var l=k+'|'+i.getAttribute('data-
referrer'),m=new Date();m.setTime(Date.now()+1000);document.cookie="x-src="+encodeURIComponent(l)+";
"+"expires="+m.toGMTString()+";path=/; domain="+window.location.hostname.replace(/^.*(\.facebook\..*)$/
i,'$1');}return i;}f.exports=h;},null);
__d('Primer',['Bootloader','CSS','ErrorUtils','Parent','clickRefAction','trackReferrer'],function
a(b,c,d,e,f,g){if(c.__markCompiled)c.__markCompiled();var h=null,i=/async(?:-post)?|dialog(?:-post)?|
theater|toggle/,j=document.documentElement;function k(r,s){r=c('Parent').byAttribute(r,s);if(!
r)return;do{var t=r.getAttribute(s);JSON.parse(t).forEach(function(u){var
v=r;c('Bootloader').loadModules.call(c('Bootloader'),[u[0]],function(w){w[u[1]](v);});});}
while(r=c('Parent').byAttribute(r.parentNode,s));return false;}
j.onclick=c('ErrorUtils').guard(function(r){r=r||window.event;h=r.target||r.srcElement;var
s=c('Parent').byTag(h,'A');if(!s)return k(h,'data-onclick');var
t=s.getAttribute('ajaxify'),u=s.href,v=t||u;if(v)c('clickRefAction')
('a',s,r).coalesce_namespace('primer');if(t&&u&&!/#$/.test(u)){var w=r.which&&r.which===2,x=r.altKey||
r.ctrlKey||r.metaKey||r.shiftKey;if(w||x)return;}var y=k(h,'data-onclick');c('trackReferrer')(s,v);var
z=s.rel&&s.rel.match(i);z=z&&z[0];switch(z){case 'dialog':case 'dialog-
post':c('Bootloader').loadModules(["AsyncDialog"],function(aa){aa.bootstrap(v,s,z);});break;case
'async':case 'async-post':c('Bootloader').loadModules(["AsyncRequest"],function(aa)
{aa.bootstrap(v,s);});break;case 'theater':c('Bootloader').loadModules(["PhotoSnowlift"],function(aa)
{aa.bootstrap(v,s);});break;case
'toggle':c('CSS').toggleClass(s.parentNode,'openToggler');c('Bootloader').loadModules(["Toggler"],functio
n(aa){aa.bootstrap(s);});break;default:return y;}return false;},'Primer
click');j.onsubmit=c('ErrorUtils').guard(function(r){r=r||window.event;var s=r.target||
r.srcElement;if(s&&s.nodeName=='FORM'&&s.getAttribute('rel')=='async'){c('clickRefAction')
```

- But the fact that it might be easy to replicate the look of a webpage leads to security issues with **phishing** attacks, where users are tricked into clicking a link to some webpage that looks legitimate but is actually controlled by an attacker, so when they put in their username and password it's actually captured.

- So going back to our simple example, we can save that code as a file on our desktop called `hello.html` [2] and open it with a browser:



- We can put some `paragraphs.html` [3] into the page too, with `<p>` tags.

```
...
<p>
Lorem ipsum ...
</p>
...
```

- We might have other tags like `<strong>`, to make part of text bold. And we can even add **attributes** to our **elements** to change them entirely:

```
...
<body bgcolor="green">
...
```

Here we've added a `bgcolor` attribute to our element of `body`, so the background color is green.

- To add a link, we'd have a tag as in `link.html` [4] that reads:

```
...
<a href=="https://www.cs50.net/">CS50</a>
```

[2] http://cdn.cs50.net/2016/mba/classes/7/src7/hello.html.src
[3] http://cdn.cs50.net/2016/mba/classes/7/src7/paragraphs.html.src
[4] http://cdn.cs50.net/2016/mba/classes/7/src7/link.html.src

```
...
```

`<a>` is an anchor (link) tag with a hyper-reference, `href`, to some address where the text is shown and the link is where the browser takes us when we click the text. But notice that we can change the text inside to not match where the link goes, and trick users if they are not paying attention to the address of the page they end up on.

- We can also include images as in `image.html`[5]:

```
...
<img alt="Grumpy Cat" src="cat.jpg"/>
...
```

Notice that we're closing this element in the same tag we used to open it, since there's nothing inside an image. And we can also put in a URL of an image from elsewhere on the internet.

- And to look up tags and how to use them, we might just Google search for "html tags" or similar.
- We might do something more advanced like a table as in `table.html`[6]:

```
...
<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>
```

[5] http://cdn.cs50.net/2016/mba/classes/7/src7/image.html.src
[6] http://cdn.cs50.net/2016/mba/classes/7/src7/table.html.src

```
    <tr>
        <td>*</td>
        <td>0</td>
        <td>#</td>
    </tr>
</table>
...
```

`tr` is a row in a table, and `td` is a cell.

- But in our webpage, we've included a background color still, and generally we want to separate our data (text) from our presentation (styles).
- We can separate our styles into the `<head>` of our page, so they end up all in one place as in `css-1.html`[7], or even in separate files like `css-2.html`[8] and `css-2.css`[9].

  # We use a line like `<link href="css-2.css" rel="stylesheet"/>` to include the CSS file, and we'd want to do this because now we can have all of our website's HTML files include this one CSS file, and update the styles on all of them at once by changing that one file.
- We take a look at Cloud 9[10], an IDE (Integrated Development Environment) that has development tools pre-installed, which Project 1 will walk us through using.
- After we log in, we notice that there is a text editor, an area that shows us what files we have, and a Terminal, which allows us to type in commands that control our virtual machine under-the-hood.
- We can start a web server with a command like `apache50 start`, which will give us a link to our website that anyone can access.
- We make a basic website that looks okay on our laptops, but has really small text on phones. We can increase the font size, which makes it more readable on phones but also really big on laptops.
- Turns out, we can make our website **responsive**, or able to resize itself depending on the screen size of the device being used. In fact, we can see in `responsive.html`[11]

[7] http://cdn.cs50.net/2016/mba/classes/7/src7/css-1.html.src
[8] http://cdn.cs50.net/2016/mba/classes/7/src7/css-2.html.src
[9] http://cdn.cs50.net/2016/mba/classes/7/src7/css-2.css.src
[10] http://c9.io
[11] http://cdn.cs50.net/2016/mba/classes/7/src7/responsive.html.src

that we can accomplish this with a `meta` tag that tells our browser to rescale our page automatically: `<meta name="viewport" content="width=device-width, initial-scale=1.0"/>`.

- We can use JavaScript code to control this with even more precision manually.
- It used to be that web developers would create separate versions for each browser since they all interpreted HTML slightly differently, since our HTML and CSS might not specify everything like margins or default font size. But now there are **libraries**, or collections of code, that other people have written that help make our websites standardized across browsers.
- So for Project 1[12], we'll be creative in being hands-on with making some webpages!

[12] https://cs50.harvard.edu/mba/projects/1/