**Week 1 Notes**

* the web is a collection of services for retrieving (largely text) content over the HTTP protocol, using the Internet as its underlying foundation.
* How does a browser work? Requests🡪Responses (including HTTP status codes)🡪Rendering (Both requests and responses can contain additional information we’ll be looking at later. e.g. User-agent)
* There are other uses for the HTTP protocol, other than simply serving up web pages
* Though some versions of HTTP allow us to keep a connection open for a little while, each independent request is just that: independent. HTTP is a stateless protocol! It doesn’t care what it did last Thursday, or yesterday, or even 11 seconds ago. If we do need state (i.e. if we want to be able to make requests that rely on stateful information), then we’ll need to provide that ourselves
* How can we host stuff ourselves? Typically, we’ll have some sort of web daemon running I That daemon will have some standard configuration file(s), to let it know where to find files to host, and how to handle particular file types F e.g. /var/www/html/ I .We can usually have additional files that give more specific instructions for what to allow/disallow (e.g. for particular individual folders) F Somewhere/somehow we’ll talk about .htaccess, .htpasswd, etc. One obvious thing to remember is that security is a thing. And you need to be mindful of what sort of access you’re granting I On a more secure system, you might not even have the option of being fast and loose with file/folder permissions
* Most browsers include some form of element inspector, as well as a JavaScript console ( These can be fantastic for troubleshooting, brainstorming and experimenting, and verifying)

**Week 2 Notes (HTML)**

* What is a web document? It’s just a plain text document. A web browser parses the document, and renders the web page. If the document says to decorate text, the browser decorates it accordingly. If the document says to use different colours, then the browser does that during the rendering process. Absolutely everything you see on a web page is either written in plain text, or included because of a request prompted by that plain text. If a web page requires an image, then the document includes an instruction to download it The page may also include instructions to download additional documents (Note: You actually can embed the image directly in a web page, by encoding it in plaintext, but this isn’t incredibly common, and is outside the scope of this course)
* (Hosting) Remember how we said ‘the web’ works: A web server holds documents A client (web browser) makes requests of documents.This means that, to have a website, you need to host the documents on a server. It isn’t terribly hard to run your own web server, but there are also plenty of hosting services available
* What is HTML? HyperText MarkupLanguage. A markup language lets you write content (text), but also include annotations within the document that indicate special ways to handle some of that text. Because we’re writing a plain text document (where the formatting is indicated by annotations within the text), the browser knows to ignore anything that might otherwise imply formatting. Extra white space (e.g. extra spaces) are automatically reduced to a single space Line breaks in the source document do not yield line breaks in the rendered page!
* (HTML Elements/Tags) The special text we include to give directives/hints to the browser are called elements (or more commonly, tags). Some have both a Start/Opening tag and an End/Closing tag Others are called empty elements (because they consist of a single tag, without a complement)
* At the very least, every document should have the following three tags: **html** — Used to indicates the boundaries of the document **head** — Used to indicate the area for style links, metadata, etc. **body** — Used to indicate the boundaries of the actual content
* There are two basic types of open/close tags: Inline, Block
* Much like modern word processors, multiple sentences together constitute a paragraph. Indicate a paragraph with the p and /p tags
* Headings are used to identify different levels of text. They employ a hierarchical naming convention.
* HTML has begun to move away from explicitly formatting text (we’ll address that task later), but there are still several tags remaining for that purpose. Note, however, that some are being replaced with semantic hints instead (even if they use the same same descriptor). **b** — bold **strong** — important **i** — italics **em** — emphasized/important **sub, sup** — Subscript/superscript **ins** — inserted text **del** — removed text **s** — no longer accurate text (note: difference is only semantics) **strike** — stop using it (as of 5) **mark** — marked/highlighted **u** (underlines, but only suggested for making soemthing ‘stylistically different’ now, like spelling errors)
* (Seperation) #<br/> - break, #<hr/>-horizontal rule
* (HTML Attributes) HTML tags also have attributes: **a** has href (as we’ll see next), All have id, style, title (for tooltips and accessibility software), class (more on this later), Most have name. Other tags have element-specific attributes, img has alt All of these attributes are ways of telling the browser to handle a tag in a very specific fashion.
* (Linking to other documents) a — anchor tag Specify target URLs with the href attribute We can specify the text to use for the anchor We can also use anchor tags for other things (including bookmarks) To link to a specific place in a document, use an anchor tag with an id, and include a reference to that id within the target URL The target attribute can let us choose to have a link open in the same window, as a pop-up, etc.
* (Images) We simply use the img tag Specify the URL of the resource via the src attribute
* (Combining elements) Note that we can often combine elements for a single purpose. e.g. a link (anchor) may have portions that are italicized, etc. If we embed an image tag within an anchor tag, we can create a clickable image for navigation.
* (Tables) Tables allow for the tabular layout of elements. There are relatively few tags to learn: **table** — the table **th** — table header **tr** — table row **td** — table data (cell) There are additional features we can include; some of which will be introduced in your lab: Borders Captions Multicolumn/multirow
* (Enumerated Types) **ul** — Unordered Lists **ol** — Ordered Lists **li** — List Item
* (Explicit Font Formatting) As the web developed and progressed, the need to separate structure from style became evident. As such, many older tags that allowed quick and easy font formatting have been either removed or deprecated. Some are still supported by most browsers, some not, but it’s advisable to avoid all of the following: **font**: color, face, size.

**u** — just for the sake of underlining. **big** (and yet, small is still just fine)

* (Generic Elements) Sometimes, we need to identify that text is special in some way, but we lack an appropriate semantic label. To address these situations, we have two generic tags where the tag itself doesn’t imply any specific meaning — we add the meaning ourselves later (e.g. with CSS). div — Block-level element. span — Inline element
* (Forms) Though it wouldn’t help much to look at them yet, forms offer a convenient mechanism for submitting information. For logins For email/message submission For transactions Pretty much anything where you need the user to type in or select information to send to a server
* (HTML5) video. Canvas. Local storage. New tags with implied usage: ( article, header, footer , nav , section , aside ,caption , figure , figcaption ,main. Extra form controls
* (FilePaths) We’ll eventually get to the point where we’ll have more complicated folder structures. It isn’t uncommon to want to separate different types of media. If we want the convenience of relative paths, but the flexibility of being able to store resources wherever, the base tag can be useful. Since it describes how to treat the document, rather than its contents, it’s used within the head tag.
* (General Design Tips) Always try to keep your layout (of the source text) as clean and readable as possible. Remember that it will automatically perform things like word wrap, and ignore line breaks, so feel free to break at any arbitrary point on each line. Make sure to line up your opening and closing tags properly. Make sure to test your site on as many browsers, devices, and sizes/aspect ratios as possible. Ensure consistent layout/interface, and give consideration to things like reasonable navigation, and storyboarding

**Week 3 (CSS)**

* What are styles? Elements within a web document have formatting rules applied to them, so the browser knows to render them differently. Styles control everything from font type and formatting to layout and positioning of elements
* (How to set Styles) There are three basic ways: 1.Per element. 2.Specified in the head of the document. 3.Specified in a separate document, and then linked from within the page
* CSS is based on the principle of setting up general rules that selectively apply to elements, depending on criteria (selectors) .Any single element might have multiple properties set to its style. An element might qualify under multiple selectors. An element might inherit from an encapsulating element. This makes sense: if you have a block of blue text, and three words of it are bolded, wouldn’t you expect those bolded words to also be blue? A single style property could potentially be applied to multiple qualifying elements Elements might be selected based on id, tag, class, or through more complicated rules
* (Styling a Document) When a browser renders a document, it has a basic hierarchy for applying styles. When a later-encountered style conflicts with an existing style, the new one replaces it. The sequence is pretty simple: 1 Browser’s Internal Style Sheet 2 User-Defined Styles 3 Webpage’s External Style Sheets 4 Webpage’s Embedded Style Sheets\* 5 Element’s Inline Styles. \*Actually, External and Embedded Style Sheets have the same priority, but this is assuming the document was written according to good conventions.
* (External Stylesheets) In our document, we’d have: **rel** indicates the relation of the link **type** indicates the document format **href** says where to find the style sheet.
* (Basic Selectors) The simplest selectors are simply the HTML tag name.

We can, however, combine selectors into groups. For example, did you notice that the table example used the same style for table, th, and td?

table,th,td {border: 1px solid black;}

We can also use a special selector \* to apply a style to any element (and by extension, all elements) within the document.

* (Contextual Selectors) We can also use selectors that acknowledge the document’s hierarchy (parent, child, sibling, descendent, etc.)
* el1 el2 – Selects all el2s inside el1s

el1>el2 – Selects all el2s that have el1 as an immediate parent

el1+el2 – Selects all el2s that are immediately after el1

el1~el2 – Selects all el2s that are preceded by a el1

All of these can help with things like, “make the links (a) inside a table green”, “make the first paragraph after a heading a different colour”, etc

* (Box Model) Elements in HTML follow a simple model: padding, border, margin. The padding is the space left between the contents of an element and its border I For a table cell, it’s the space between the text and the border of the cell The border is just that: a border I Borders also have additional attributes. e.g. style The margin is the space reserved outside the element I For example, suppose you wanted all images to stay separate from the surrounding text. You could set a large margin, and it would require a gap around all sides of the images. These spacings may be set for all four sides, or individually See more information here: http://www.w3.org/TR/CSS2/box.html <http://www.w3schools.com/cssref/default.asp>
* (Sizing) Elements may have their sizes explicitly set

You can set width, height, or both

This could cause issues with overflow

Sizes may be set as absolute values, or percentages

Both can be good or bad, depending on the situation

It’s often good to use a mix

The most flexible content should be the percentage (good for elastic layouts)

This is one of the uses of em

* (Floating elements) All elements have a float property. The default is none.

Floating an element breaks it from the normal text flow by attaching it to the left or right side of the enclosing element

e.g. Float an image to the right to let text wrap around/past it on its left

Note that you can also use clear to force an element to come after the floated element

* (Positioning) Every element has a position. The default is static.

static – Normal rendering; elements appear in the document when they appear in the source

absolute – Element is positioned relative the first non-static ancestor element

fixed – Element is positioned relative browser window

relative – Element is positioned relative where it would normally appear

Changing the position doesn’t really do much until you also set at least one of top, bottom, left, or right. By default, they’re set to auto, but you can specify either a distance or a percentage.

* (Z-index) The z-index property lets you assign a layer priority. Basically, think of it as being like a depth (elements with a higher z-index are on top of those with a lower z-index)
* (More advanced selectors) Though individual spans could be formatted with inline styles, this is a perfect time to bring up the idea of a class.

A class is a label that you define

It may be used to define your own semantic meaning

It may be applied to any element you wish

You can set up CSS rules for your class

It simply follows the pattern of .class

You can combine with the other tricks. e.g. p .class

* (Attribute Selectors) Similar to classes, you can create a rule unique to a specific element (identified by its id)

An id should be unique in a document

Set the CSS rule via: #id property: value;

So, why not just use an inline style then? Even for single-use styles, it’s better to keep all style information together, so you don’t have to manually find it within the document.

You can also select elements based on whether or not they’ve set a particular attribute (e.g. href), or even the precise value of that attribute.

* (Psuedo Classes and Elements) CSS also lets you set properties based on temporary conditions or hierarchical positioning. These are not true classes, but are included for convenience. For example: Being able to modify the first character of a paragraph Specifying the formatting of an element while the mouse is hovering over it.

<https://www.w3schools.com/cssref/css_selectors.asp>

* (Visibility) The visibility of any element can be set via the visibility property (visible, hidden, or collapse). Two obvious question: Why? How is this different from display: none? Note that collapse isn’t the same thing as being collapsible. That’s... a bit trickier
* (Additional Notes) When setting any property,

property: value !important;

prevents later selectors/style rules from replacing this rule. It is especially vital for those with visual impairments.

<https://www.w3schools.com/css/css_howto.asp>

<https://www.w3schools.com/cssref/tryit.asp?filename=trycss_cursor>

**Week 4/5 (JavaScript)**

* JavaScript is a client-side interpreted language
* Several uses for scripting:

Changing the styles of elements

Storing/retrieving client-side data

Connecting to (and communicating with) a server

Altering the contents of the document tree (etc. etc.)

* We have several ways to run JavaScript in the browser

Within the developer’s console

Embedded in the page

From a separate file linked by the page

Within user scripts, extensions, etc.

* (Variables) JavaScript is a weakly, dynamicaly-typed language. We can assign (and reassign) variables arbitrarily. We can actually just do an assignment without declaring anything, but sometimes we will want to declare one with the var keyword
* (Booleans and Conditions) We do have a true and false, but several other things can work for conditionals Numbers: 0 for false; anything else for true Strings (other than the empty string) resolve to true Arrays, objects, etc. (while null is false)
* (Functions) Of course, with just about any language, procedural abstraction is a requirement. JavaScript is no exception.

Simply use the function keyword to start declaring a function, and everything else is the same as with most languages. You don’t need to declare ahead of time if you might want to return anything

If desired, you can nest functions .Of course, the same scope rules apply here as well

JavaScript does support variadic functions. If the final parameter starts with ..., additional arguments get put into the a single array by that name

Functions can effectively be treated like values. Assign them to variables, or pass them as arguments

There’s an alternate syntax available for declaring

e.g. dealie=(a,b)=>{return a\*b;}

This can be more useful than it seems F e.g. (random=>4)() F (actual useful info here: https://developer.mozilla.org/en-US/ docs/Web/JavaScript/Reference/Functions/Arrow\_functions )

* (How to get JS into a page) Use the script HTML tag to list the JavaScript to run

The script tag can have a src attribute set to specify a separate .js file, instead

* (Arrays) We know what an array is: an ordered collection of... things.

Unlike with some languages, JavaScript arrays don’t need to be homogeneous

We have a few ways to declare them, including forming them with square brackets, and using the Array constructor

We can assign to arbitrary indices, but be careful with that. Let’s try out a couple examples? Though we can assign values by using other things as indices (e.g. strings), this really isn’t the same thing, so be exceptionally careful

* (For-each loops) Let’s say I want to iterate over an array. We have a variant of the for loop for this! However, be careful: we’re actually iterating over the keys, not the values Also, don’t rely on the sequence necessarily predictable. It probably will be, but the language specification doesn’t require it I Just traverse it traditionally if you need to enforce a sequence
* Arrays have a built in .sort function
* (Accessing the document’s contents) You have two basic objects that define the starting point of many operations: document and window

Technically, document is a property of window

When we’re inspecting elements in the browser, we’re basically looking at the document

We can get into more interesting uses of the DOM later, but for now let’s have some fun Basically, we’re going to be asking the document for references to nodes within the tree. Once we’re at a given node, we can then follow along to children, siblings, and even its parent.

* (Modifying the tree directly) If desired, we can: Remove an existing element (or subtree!) from the tree

Create a whole new element programmatically, and insert it wherever we want.

By inserting an existing element into a new location, reposition members

* (Hoisting) JavaScript normally does something called hoisting: pulling up declarations, potentially before they’re used.

It’s worth noting that initialized variables will not be hoisted. On the other hand, functions are. d e a l i e ( ) ;

f u n c t i o n d e a l i e ( ) { c o n s o l e . l o g ( ’ nea t ’ ) ; } This is perfectly legal.

If you want to curb such shenanigans, strict mode is a reasonable solution. Just start the first block with "use strict"; and it’ll force you to actually declare things before using them.

* (Objects) Objects in JavaScript are a highly dynamic data structure. They’re little more than dictionaries, with listings of the member attributes and methods.

If desired, we can create one almost trivially: x={val:3};

This is known as an object literal: defined upon first use

You store things as property:value pairs

Technically, you can use a string as the property label, but be very careful with this!

e.g. x={"y z":"monkeys"} is legal, but tedious

Normally access members via the dot notation (though bracket notation also works) Technically, you can iterate over an object as well

* JavsScript is dynamically typed
* (JSON) Before we can get to communicating with a server later, we’ll first need to know a couple ways to bundle up data to make it easier to both access and transmit/receive.

A popular format, not just for JavaScript but for several languages, is the JavaScript Object Notation format. Basically:

We encapsulate blocks with curly braces

We can have as many (comma-separated) property:value pairs as we like (with no expected sequence with regards to sibling tuples))

We can define lists of entries with square brackets. JSON is literally just javascript objects

* (First Class Objects) Functions are first-class objects (and, uh, objects). We’ve already seen evidence of this when we assigned them to variables, etc. Of equal (greater?) importance is that we can also use function references as parameters.

This lies at the heart of callbacks. (Also, because they’re objects, you can assign additional members to them, which is neat)

Semi on-topic, look at this fascinating answer here: https://stackoverflow.com/questions/48923021/ javascript-local-static-variable#answer-50033681

* (Event Handling) We’ve already seen a few trivial events (e.g. onclick), but it was pretty limited. We didn’t have many events available to us as HTML attributes, we couldn’t have one click trigger multiple actions, nor could we have multiple triggers for the same action (as easily as we might like).

The solution is simple: proper event handling is based on the Observer design pattern.

To declare our interest in observing some element, we associate an event listener with it

There are numerous mouse, keyboard, touch, etc. events we can associate (as well as creating custom events)

We could easily have a whole lecture just on events, but some really interesting reading: https://www.w3schools.com/jsref/obj\_events.asp Major event types

https://javascript.info/mouse-drag-and-drop Great drag-n-drop example with explanations

https://javascript.info/onscroll I A scrolling demo (consider those ’infinitely-scrolling’ pages with endless content?)

(Storing informational locally) Though cookies are great for storing information for the server, sometimes you need to store data for yourself.

JavaScript provides two special objects for this: localStorage and sessionStorage.

Howsabout we try modifying that page so it can save our notes for us?

We store in web storage by using setItem, with a property name to associate with the data

Only store strings (JSON.stringify’s lookin’ useful now, eh?)

What are some considerations? We can definitely save/load on button presses, but do we want to rely on memory? Could we save on keypress? Is that reasonable? What’s the alternative? IWe also might like window.onbeforeunload

**Week 6 (CGI-PHP)**

* There are lots of reasons we’ll sometimes need to script things on the server side, rather than the client side.

Some content must be dynamically generated

Content might be generated based on external data

Pages could have content specific to individual users etc. etc.

Generally speaking, these scripting languages will construct the document that is sent to the browser.

* (Common Gateway Interface) Assume the following requirements of a server-side script:

The ability to reuse HTML text that’s been written once, across multiple pages

Perform substitutions on portions of written text

Make decisions based on server- or client-side data, to selectively serve up extra content Receive text-based requests, and prepend additional text-based metadatas

What do these all have in common? They’re really all about handling text.

Think about how you’d handle such text processing outside of a web environment.

You might write a simple command-line program, right? That isn’t too far from the actual solution.

CGI behaves approximately like a command-line script (receiving text, and printing text back to the remote user)

There’s some additional overhead for things like making form data available, etc.

The CGI-enabled server simply runs these scripts, and then returns their output to the requesting browser

There are several supported languages, but today we’ll be using PHP

* (Strings) <https://www.w3schools.com/php/php_ref_string.asp>

<http://php.net/manual/en/ref.strings.php>

* (Arrays) By now, we know what an array is; we just need to know how to array.

We create them via the array function

There’s a special function, print r, for printing the rows of an array

We also have associative arrays available to us!

The foreach loop can be pretty convenient for arrays, since it doesn’t care about gaps in the indices

We do have multidimensional arrays available, simply as arrays of arrays.

* (Functions) In the example for the last slide, did you notice how many times I had the exact same code, just to produce a newline? If only we’d had some means of abstracting that behaviour!

f u n c t i o n b r e a k i t ( $ l i n e s =1) {

f o r ( $ i =0;$i<lines;$i++)

echo “<br/>\n”;

return “neat!”;

Unlike variable names, function names are not case-sensitive!

* (Scope) Now’s as good a time as any to discuss scope.

Variables declared at the file-level are considered to have global scope

Variables declared within functions have local scope

Weird quirk: global variables need to be explicitly added to functions, via the global keyword

We also have static variables, which behave the same as in C/C++, but hopefully won’t be needing those! Additionally, we can take a look at our first superglobal, $GLOBALS

* (Cookies) We’re finally getting somewhere interesting! Remember the capture from the beginning? When the server sends a web document to a browser, it can also include (in the header) any number of cookies. PHP can both assign and access cookies.

As a reminder, a server doesn’t request cookies; the browser sends them preemptively with each request. We’re simply able to access the information that’s already been given by the client

To access cookies, just use $ COOKIE. (Set them with setcookie) Because of how cookies are passed along, consider what restrictions that might entail.

* (Sessions) Very similar to the idea of cookies are sessions. Let’s take a look at the example first, and then see if we can work out a few questions: Where is this data stored? Can the browser access these values? Does the browser store anything? How long does the data last for?

Sometimes, we need the browser to provide additional information to the server. But... how? There are actually a few ways, but two we’ll talk about now: GET and POST

Of course, these refer to two of the fundamental operators available in HTTP

In PHP, we have $ GET and $ POST superglobal arrays. We also have $ REQUEST, but that’s just an amalgamation of $ GET, $ POST, and $ COOKIE (in theory) We’ll look at POST in just a moment, but let’s look at GET for now.

Possibly useful functions: htmlspecialchars/htmlentities, and urlencode/urldecode

Also, perhaps we should have a discussion of when to use (or not use) GET, as well as idempotency?

**Week 7 (AJAX)**

* In previous weeks, we’ve discussed how to write scripts on a server to construt or serve web content, and how to script events within the page to be executed by the browser.

Many pages can be written to simply rely on page reloads for new content, but oftentimes we’d prefer to have dynamic access without needing to refresh the whole page.

For this, we just open a connection to a server-side script, and make a request (possibly passing information along, and typically receiving something back

One catch is that there’s often a slight delay in receiving the response

Colloquially, any page operating on this basic principle is usually referred to as using Ajax.

Let’s start with the simplest example: our page wants to request a random number from the server.

The server-side script will just return a value (including MIME type)

The page will need to create a new XMLHttpRequest object, and send the request. If we wanted, we could add extra information to it first. We set the callback (function to invoke) for when the page is done

Assuming the page was retrieved without error, we use its data

We often call this asynchronous communication, because the tail end of it happens... whenever.

Naturally, our first example was a bit simple. Typically, there are two uses for this technique:

For frequent checks for updates (polling). Today, you might replace these use cases with web sockets or the push API. There are still plenty of inboxes that have, for example, a setInterval to ‘phone home’

Dynamic responses to user input .Saving changes to the server for an editor . Requesting new content (e.g. ‘infinite scrolling’ browsing)

* (Promises) Technically, we don’t need to use the XMLHttpRequest object; we could fetch pages instead, but there’s something different with how they work...

fetch is part of a new suite of modern JavaScript tools, designed around better coding conventions. In particular, rather than the option of performing asynchronously, it’s designed around it

It makes use of promises So, what’s a promise? Simply put, a mechanism for attaching callbacks

* (htaccess) For your web server daemon, you’ll typically have a .conf file somewhere dictating all of the default configuration options. However, you can override these. This includes creating exceptions: specific folders that operate on different rules

Of course, the conf file may permanently restrict some options

We set these rules in the .htaccess file. Simply include one in a folder, and it’ll apply there (and usually any subfolder from there as well, unless you have a deeper .htaccess to undo it).

* (Index pages) Lots of websites you go to don’t require a specific filename at the end of the URL, right? (e.g. you go to https://google.com, not https://google.com/findsomething.cgi)

We achieve that by specifying one or more default DirectoryIndex files

Basically, if any of those files exist within the folder, they’ll be sent when no filename is included.

* (Security) Generally, the best security is actual security. However, one way to discourage people from poking around is to simply not let them access anything at all. IP filters are one way to do this. Typically, there’ll be two parts:

Who can connect

Who can’t

Also, you’ll want to specify the precedence of the two (e.g. ’disallow all, but make an exception for this one’).

When we get to databases in a couple weeks, we’ll briefly discuss proper authentication. However, sometimes you just need to restrict access to resources, without substantial concern for scalability, user accounts, etc. We do have something available to us that provides hardcoded password protection for folders. It’s the .htpasswd file At their simplest, all a .htpasswd file is is a list of entries of:

Username

A hash of the password

Maybe extras (e.g. salt)

We can generate it via the htpasswd command.

* (Hotlinking) Have you ever seen a neat picture, and tried including it in a link to someone to share, but the other person couldn’t open it? And you re-check the link, and it’s still showing up just fine for you? This is particularly common if you’re trying to embed the image into a page, etc.

People tend to not like having their bandwidth wasted for things other than the actual usage of their sites

Making a direct link to someone else’s assets is called hotlinking

The common form of discouraging hotlinks is to check the page that referred the browser to that asset (by inspecting the HTTP header). If desired, you can specify an alternate resource to give such users

* (MIME types) Suppose you had imagemagick installed for your web server (ours currently doesn’t). You could generate an image via a .php script! But, it isn’t as fun if it looks like it’s still a script (thanks to the extension). If you want, you can actually define alternate behaviours to associate with file extensions.

e.g. .png files are now PHP scripts

We won’t do an example of this, as it’s most commonly for either images or hiding .cgi/.php extensions. As a silly use case, you could have dynamically-generated CSS files, that still appear to have the .css extension. (We could also tell it to not execute script files, to make it easier to host examples, but... nah!) http://www.htaccess-guide.com/adding-mime-types/ <http://www.htaccess-guide.com/forcing-scripts-to-display-as-source-code/>

**Week 8 (CGI/Python)**

* Remember that server-side scripting doesn’t have to be PHP. CGI relies on basic IO streams, so lots of languages are viable
* Before we can get to the CGI side, we’ll need a basic familiarity with Python. It’s a dynamically-typed interpreted language, and has plenty of similarities with JavaScript.
* To create a Python script, it’s basically the same process as making a bash or PHP script On sandcastle, we’ll be using the shebang #/usr/bin/python If you want to run it from the command-line, you can get away with a chmod 700 For Python, the typical extension is .py, though we’ll be switching to .cgi later
* (Cookies/Sessions) Assigning cookies simply requires including additional headers at the top. Good cookie-readin’:

https://jayconrod.com/posts/17/ how-to-use-http-cookies-in-python https://www.tutorialspoint.com/ How-to-retrieve-cookies-in-Python-CGI-Programming

Also, cgi readin’: <https://docs.python.org/2/library/cgi.html>

But what about sessions? Those are... more complicated.

Recall how PHP handled sessions

We can’t directly share sessions between the two languages

Vanilla Python doesn’t include anything for them (you’re stuck either simulating with files/databases, or using a framework)

Neat: <http://cgi.tutorial.codepoint.net/a-session-class>

(Additional Info) It’s not just PHP; ’sessions’ generally won’t share across languages because their data is stored in different places

e.g. PHP held everything internally, with only the cookie externally-facing

Traditionally, there are a few ways to handle this: I Share access to same database (unfortunately not an option for us) I Using a ’driver’, or inter-process communication I One script invoking or retrieving another (we have exec and curl, but it’s preferable to avoid, both for performance and security reasons)

In practice, if we end up needing this, we can fall back on calling each other (if absolutely necessary), but for this course we’re probably best off either sticking to a single langauge at a time, distributing information across them (e.g. document structure in one, database in another, maybe with JavaScript to glue it together)

**Week 9 (Mongo)**

* What we want is a more flexible (and ideally, efficient) means of storing, searching, modifying, and retrieving content. In short: a database
* When it comes to web design, the database server runs as a process somewhere, and server-side scripts query it. Even if your client code needs to look up a single trivial piece of data, it’s still abstracted behind server-side code, for security (as well as other reasons) This means we need to connect to a database server before we can access its contents.
* (How to get dynamic content) I might need to switch to the board for this one, but the basic idea:

A page uses JavaScript to send an asynchronous request to the server (e.g. PHP, Python, etc.) That script will commonly tie into a shared common snippet of code that handles data management e.g. in PHP, include a script that make an object to connect to the database

When the script needs data, it constructs a query and passes it along through the data management

The response from the data handler might be more than you need, so parse and trim Return what’s needed

* (Kinds of databases) There are two common basic supercategories of database: relational and... other

A relational database will typically involve a well-defined schema to store data I Data is arranged in tables, with interconnections, etc. I A declarative language is used to query (e.g. SQL)

The umbrella term of NoSQL refers to everything else I e.g. documents, graphs, tables, etc. I Many are designed with specific use cases in mind I Many focus on other features, rather than efficient queries or trigger

* (Mongo Client)

db . p e o p l e . i n s e r tM a n y ( [

{

name : ’ Abby ’ , age : 3 7 , f o o d s : [ ’ p i z z a ’ , ’ cake ’ ]

} ,

{

name : ’ Bob ’ , age : 4 5 , f o o d s : [ ’ c h e e t o s ’ , ’ salmon ’ , ’ t a c o s ’ ]

} ,

{

name : ’ Stan ’ , age : 6 6 6 , f o o d s : [ ’ s o u l s ’ , ’ s u f f e r i n g ’ , ’ p i e r o g i ’ ]

}

] )

When it comes to MongoDB, each of these JSON objects is called a document, so whenever we talk about a document, that’s what we mean. This is important to understand, because we don’t only mean records: we’ll be writing our queries in a JSON format, so those are also documents.

* (Basic Queries) Let’s start with the simplest query possible: db.people.find()

You’ll see that we basically get everything (and also that a unique identifier is automagically assigned to each document).

The find function actually takes two parameters: query and projection.

db.people.find({},{}) yields the same result.

The projection is just which fields you want to receive I e.g. db.people.find({},{name:true}) will yield the names F And the Object IDs, though you can explicitly refuse those

The query is how it decides which entries to include at all I e.g. db.people.find({name:’Bob’}) will find every record with a name of Bob I db.people.find({age:{$lt:100}}) gives records for users less than a century old

<https://docs.mongodb.com/manual/reference/operator/query/>

<https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/>

<https://docs.mongodb.com/manual/reference/mongo-shell/>

* (Python) As mentioned earlier, we have a pymongo module. And we’ll still need to authenticate before use.

There are a few differences from the shell, including:

Some function names have changed I e.g. findOne is find one

It uses dictionaries instead of JSONs I e.g. {’age’:{’$lt’:100}} instead of {age:{$lt:100}}

The cursor behaves slightly differently in the shell, but we’d have to account for that for a CGI script anyhoo

* (Sessions) Remember what we’ll need: 1 A cookie that connects a user to a session 2 A persistent entry that stores data for the session 3 The session data to be automatically cleared when it’s no longer needed
* (Security) Note that there is still a security concern regarding how we’ve set up our scripts. Sticking your credentials right into code is sometimes done, but really shouldn’t be. Basically, if someone were to gain access to your system, he or she could easily read your password.

At the very least, be very careful with your file permissions (chmod)

Consider creating a single file to build/retrieve the authenticated Mongo Client, so at least your credentials are only in a single script

There are other forms of authentication, including certificate-based

Generally speaking, the more sensitive the information, the more hoops you should need to jump through to access anything

<https://www.w3schools.com/python/python_mongodb_insert.asp>

<https://docs.mongodb.com/manual/reference/mongo-shell/>

<https://docs.mongodb.com/manual/tutorial/query-documents/>

<https://docs.mongodb.com/manual/reference/operator/query/>

<https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/>

<https://docs.mongodb.com/manual/reference/operator/update/>

**Security (Week 10)**

* (How to protect passwords) A hash is a one-way mathematical transformation of some data into a fixed-length chunk

Though very difficult to reverse, and seemingly random-ish, it’s actually deterministic and repeatable

So you could just store the result of such a transformation. When the user reproduces that transformation (using a possible password), compare that against the stored version.

Rainbow (and basic lookup) tables rely on the concept of doing work once, and then using that work indefinitely.

A collision in a hash function is when two values are hashed to the same result. They aren’t terribly common

That means when you see the same hash multiple times, it probably means they’re the same password I (And you can use the same password, regardless)

A salt is a small string of data you concatenate to the password (or hashed password) before hashing, so the same password twice will generate two results (so long as the salts differ)

* (Challenges) When you present an additional barrier to authentication, we call that a challenge. These tend to either confirm you aren’t trying to automate, discourage replays, or even require additional identification. e.g.:

A captcha

Including a nonce (or time-limited component) in the login form

2FA

Verifying unrecognized devices

* (Logging not just for debugging) What should we be tracking, both across sessions and within a single session? e.g.

IP address (or addresses)

User agent string, or anythign else you can track or derive a signature from

Why? Let’s talk about session hijacking, and how it relates to the above

* (HTTP Strict Transport Security) HSTS, in technical terms, waits beside a browser waiting for the browser to try visiting the insecure version of your site. When that happens, it smacks your browser over the head with a sock stuffed with nickels.

Less formally, when a user accesses your page, there’s an optional HTTP header you can include for Strict-Transport-Security (along with an expiry time in seconds, and possibly extras about subdomains)

When the browser tries to return to a site with HSTS turned on, it won’t accept the insecure version Yeah, but that’s for when the user returns to the site. What about the first visit? Let’s have a chat about whitelists...

There are a few things you’ll quickly encounter if you try starting a new website: People (or bots) will try to hack you People will try to take down your site People will try to hack you People will try to inject code People will try to hack you People will try to extract information or assets People will try to hack you No matter which you’re experiencing at the moment, the result may be the same: there’s no resources left to actually run your site.

What sorts of threats do we need to be vigilant for? DoS (and DDoS) I Note that it can be very difficult to distinguish between a DDoS and, for example, the ’reddit hug of death’ I Back in the day, search engines were to be feared

Attempts to gain access to administrative accounts

Cross-site scripting attacks

Scraping links to assets for hotlinking/hijacking Basically, others could either intentionally harm your site, or unintentionally exploit it to the point of breaking.

What can we do?If you keep seeing the same IP address, you might block/throttle it, but that doesn’t help much for a DDoS

Of course, failed login attempts from the same browser/IP can result in a temporary block I Fail2ban is interesting software

Sometimes the only solution is overprovisioning/elastic load balancing I Basically outspend the attackers

Handling failed login attempts from different IP addresses can be a bit trickier, though.

Make sure to limit your liabilities by taking basic precautions, from a systems design perspective:

Whenever possible, isolate systems I e.g. if you have an email system only to be used within an office, don’t add a firewall exception for it

Remove obsolete pages you never intend to use again I Suppose you migrate from an old page to a new system. Are you still remembering to install updates for... that thing you never use? Nah, I’m sure it’s always the first thing you’d think of, right?

Be very cognizant of which services are (and should be) externally-facing

A surprisingly large number of sites get compromised, and hten used as platforms for attacks on others.

There’s only one way to even suspect your site is safe, and it’s closely analogous to how you know your programs are free of bugs: You need to break it. (Or try)

Penetration testing is the act of searching out vulnerabilities and exploits within a system, to gauge how susceptible it is to outside attack

There are pentest distributions and frameworks (kali, metasploit, etc.)

If your site starts becoming large, or is important to your livelihood, it might be worth considering hiring an external consultant

**COSC 2P89 – Lab 5 – PHP** This week, we'll be focusing on server-side scripting If you happen to have what you made for your previous labs handy, it might help speed one part up a bit. Tasks: For this lab, all I want is a single page with the following: • The first time (and only the first time) you go to the page, display something unique (e.g. legal stuff) • Assemble the page from more than one file • For the full 4 marks (as opposed to 3), have the ability to enter some information onto this page (e.g. a nickname) that changes the display for subsequent loads Our first page (anyone else feeling déja vu?) We'll start by making a very simple (but still partially-scripted) webpage. We're going to care about permissions, so you'll need a terminal (e.g. PuTTY, or simply booting up under Linux) up for at least part of this. Of course, since we'll be running scripts on sandcastle, unlike with the earlier labs, you'll need to be hosting everything we do on your public\_html. So, create a folder in there (or use the same as before; I'm not the boss of you). In that folder, create a file. I'll call mine simple.php. Its contents will be pretty trivial: #!/usr/bin/php-cgi

Hello!  
How about you?

I know you know this from lecture, but as a reminder: • The first line is a shebang: a comment telling the environment where to find the executable to run scripts • There's a bit of HTML. Note that we don't do anything special for that • We start a block of script with • We can use either echo or print for displaying text (with only minimal differences between them) • We can mark strings with " or ', though the latter tends to be a little more literal Let's try get it running. If you were to go to its address right now in a browser, you'd get a “500 Internal Server Error”1 \*. That's because of how PHP/CGI scripts run. The default permissions are 600. The 00 portion is just fine. It simply says no one else is allowed to use the script at all. The 6 means you give yourself (i.e. the 'owner') permission to read, and to write changes to it, but not execute it. Let's change that last part. In the terminal, navigate to that folder, and type: chmod 700 simple.php Naturally, if you chose a different filename, use that. Try viewing the page in your browser, and it should render correctly now. 1 Well, I do. Default permissions could be different for you. By the way, in case you're wondering, those permissions do make sense, so long as you know what's going on: • Even though 'other people' could view your page, there aren't 'other people' accessing the file, in the sense of other accounts • Rather, the server is becoming you to run the script In your browser, try right-clicking and choose 'View source'. Two things worth noticing: 1. You don't see the shebang there. The server doesn't forward that along 2. The \n did create a newline, but that doesn't really mean anything in HTML So, we're good so far. If you don't want to keep going back to the terminal for permissions, you can just copy the existing script, which should preserve the permissions on the new files. Scripting a page While although we can eventually get to more complicated behaviours, most of our PHP scripting will be about generating or assembling pages. Let's try a few attempts at the former. For this one, I'll call it dynamic.php. I'll start very small: #!/usr/bin/php-cgi Broccoli University. May contain trace amounts of Cauliflower University ---body.html---

Content unique to *this* page goes here.

---grandstyle.css--- /\*The permissions can matter for this one! Try 644?\*/ body {background-color:grey;} p {background-color:red;} ---subdomainstyle.css--- header {background-color:cyan;} footer {background-color:green;} Okay, so now we have four pieces of a webpage, but no actual page. Let's fix that: ---combined.php--- #!/usr/bin/php-cgi

In this case, combined.php is “the webpage”; everything else is just supplementary material. (In a more 'real' example, you might have other folders in play, possibly with different permissions, etc.) There's just one part I don't quite like. Those style sheets… The premise here is that the grandstyle would be across the entire site, and the subdomainstyle would be for a collection of pages. Think Brock vs COSC: still plenty of pages reusing the same content, but one implies further specialization. So external style sheets makes sense, but it's unfortunate to have to keep explicitly adding them to each page, right? Is there perhaps a better version? Feel free to skip this part if you like. It's just a refresher on looping over an array (with all of two entries). Let's just look at a possible alternate implementation: #!/usr/bin/php-cgi

Side note: this was an example of having PHP before the HTML even started. There are actually lots of reasons to do that, including (but not limited to): • Defining cookies • Deciding whether or not to redirect (or grant access) • Handling received form values • Defining filenames, constants, etc. in an easily-accessible location • Initializing data sources (e.g. databases, objects, files, etc.) C is for Cookie; I already made this joke... Let's talk about cookies. As you'll recall, cookies are small snippets of text the server can ask a browser to hang onto (and return in the header of future page requests). They're pretty simple, but potentially very powerful. Since we aren't dealing with databases yet, we won't have much to tie to them, but it's still a good time to get started with the basics. Let's start with those irritating disclaimers lots of websites have now. We could program in some form of agreeing to the terms, etc. But most just go by the, “if you continue, you're agreeing” routine, so we really just need to display a message the first time a user visits, and never again. This kind of thing is great to split off into a separate php file, because it keeps the bookkeeping away from the actual content. However, because it's so simple, I won't be doing that here. First, we need to check if the cookie already exists. What will we call it? How about disclaimer? Up at the top, we could try something like: $disclaimer=isset($\_COOKIE['disclaimer'])?"":"nertz"; Obviously it's just a placeholder. • $\_COOKIE is another superglobal, which contains all the cookies we were given by the browser • isset is a function that returns a boolean for whether or not the requested term is defined So, what actual message might we use? How about a display style? $disclaimer=isset($\_COOKIE['disclaimer'])?"none":"block"; The idea is we could stick a disclaimer somewhere in the page, but selectively hide or display it. The disclaimer (embedded somewhere else on the page) might look like: echo <<

You know you could turn off cookies in your browser, right?

TERM; One problem: it shows up every time. That's because we haven't assigned a cookie yet. We'll do that after the line where we check the current state of the cookie, but before we start the tag. setcookie('disclaimer','sure',time()+60\*60\*2,"/","www.cosc.brocku.ca"); You can read more about usage here: http://php.net/manual/en/function.setcookie.php Basically, we're telling it to create the cookie, assign a value of sure, set the expiry to two hours in the future, make it accessible to all file paths within the subdomain, and use a subdomain of COSC. If you need to check out cookies for debugging purposes (and don't want to use the browser's inspector), don't forget that print\_r is a thing: print\_r($\_COOKIE); On platonic sessions… I mean forms We already covered sessions in lecture, but they're handy enough to revisit. Cookies put everything into the browser, which means they're under user control. Sometimes, you'd like to have server-side information, and only use a cookie to tie it to a user. If we wanted to, we could set this up ourselves, by generating a unique ID for a user, sending that ID in a cookie, and then storing the rest of the information in either a text file or a database on the server side. Except… PHP can already do that for us. Let's say, for example, that you'd just like a little bit of customization. Maybe a preferred CSS theme, or a server-side temporary shopping cart (before they log in). A session can be great for that. However, we'll go even simpler: I want to specify a nickname. I'll start by creating my file. I'll go with sessform.php. The first step is to start the session. When you tell PHP to start a session, it either retrieves the stored information associated with the user's session id cookie — if one exists — or it creates a new entry. #!/usr/bin/php-cgi

As you can see, we have another superglobal: $\_SESSION. How do we assign the nickname? Well, it's an array. So we just use it like one: #!/usr/bin/php-cgi

Does this actually make sense? Is it fine to assign new session values so low in the script? • With cookies, it's normally safest to assign new values ahead of the document, since cookie values are sent in the HTTP header • But with sessions, the only thing the browser needs to store is the PHP session ID, meaning you only need to keep session\_start() at the top. Everything else can happen wherever you like Of course, the new Nickname won't show up until you reload the page. But that's kind of silly, right? Giving everyone the same nickname? What we'd like to do is retrieve some information from the user. Like… a form? Forms have two parts: a collection of HTML tags for the form itself, and its various input elements; and superglobals for accessing any form information that the browser has sent to the script. We'll do the simplest form possible: #!/usr/bin/php-cgi

Top of Form

So, what do folks call ya? 

Bottom of Form

• The action is the address to submit the data to ◦ It follows the same convention as anchor href's, so this example says “the same page” • The method is get or post ◦ There's no practical reason to submit a form via get • If you omit the value from the submit (button), it'll still work • We typically use the name attribute for form elements Now let's just modify the page to make use of this form information! We only need to modify the top bit: Reload the page and give it a try. Whenever you come back (assuming you don't close the window or lose the session), it remembers your nickname. If you wanted, you could even remove that form when it already has a name… (did you notice a problem?) ((seriously, try it out. What's the problem?)) Did you try reloading the page after submitting? It gives one of those annoying, “I'mma resubmit everything, because I assume that's what you meant by trying to reload the page instead of clicking the 'submit' button” messages. sigh. This is actually easy to fix: • The problem is partially exacerbated by the address (action) being # ◦ Because it thinks it's always a GET submission ◦ We can mildly improve it by explicitly stating the action as sessform.php, but we'll still have the issue when trying to reload from a POST • Forms normally submit to a different address, which will then often automatically redirect you to another page • We can just 'redirect' to… the same place #!/usr/bin/php-cgi

Top of Form

So, what do folks call ya? 

Bottom of Form

Now, even though it ostensibly takes us to sessform.php#, we still end up on sessform.php. What else could we do? Well, it's a whole new language, right? There's tons we haven't covered: • PHP is quite capable for math, but it isn't uncommon to create webpages with minimal server-side math • We could have tried creating (or at least reading) text files, but that tends to not play as nicely with concurrency, so it's often of limited value • We never made use of the browser's user agent string ◦ By all means, feel free to take a look at how it changes, depending on whether you set the developer tools in the browser to mimic a mobile device (responsive design mode) What about the actual submission task? Well, all you need to do is to combine the three separate concepts we've covered. My suggestion would be to take your nickname example from above, and then add in an include for a separate file for the cookie disclaimer. Of course, it's up to you whether you want to use $\_COOKIE for that, or just use a session. Either way, it'll be cookie-based, right? Addendum Depending on how you're editing your scripts, a problem may arise. Though PHP is written in plaintext, that term actually encompasses a few different variations. In specific, PHP assumes lines will be terminated with a linefeed, rather than the Windows convention of CR/LF. The point? • You can edit your scripts with nano/pico • You can use Notepad++ if you change the line endings to LF • You can probably use Visual Studio Code • You will not want to use Notepad Alternatively, even if what you're writing is in the wrong format, there's a server-side command that can easily fix it for you. Suppose your script was dealie.php: dos2unix dealie.php If you're going to use an editor that isn't compatible (e.g. Notepad), just re-run that after any time you save.

**COSC 2P89 – Lab 6 – Ajax** This week, we'll have our client-side scripts communicate with server-side scripts You shouldn't need what you wrote for past weeks for this one, but it's up to you. Tasks: For this lab, all you need do is complete the final task: • A 'tabbed' display • Click a tab to display the corresponding listing • Click other tabs to switch the display to the new selections Asynchronous requests We'll only be using asynchronous requests because they're vastly more appropriate for virtually every use case. As you'll recall from lecture, we have two common approaches to initiating an HTTP request: fetch, and XMLHttpRequest. Fetching is the newer technique, and is centred around the idea of Promises. They're pretty powerful, allow you to easily specify how to handle success, failure, etc., and they fit in well with more modern programming conventions. They're also… a wee bit harder to read and write initially. Similarly, XMLHttpRequests are (mostly) older1 , and a bit more traditional, syntactically-speaking. Considering they're both roughly comparable, you're welcome to use whichever you prefer; our first example will have a simple fetch, after which I'm partial to old-school. Let's start by loading a simple page, shall we? Technically, we could load an HTML page, but that'd feel silly. Let's make a really quick PHP script: --favouritepies.php-- #!/usr/bin/php-cgi

If you ask me, pie is the best! truth it!

Let's parse out what's going on: • The fetch creates a Promise object ◦ Think of a 'promise' as planning ahead for some future event you know will happen ◦ Initially, it's in a pending state ◦ Everything you put inside a .then doesn't happen immediately • Once that promise resolves (the fetch is complete), it invokes the callback you provided it ◦ You could use the response to check its status, etc., but that isn't really needed this time ◦ The default response type is text, but you could change that ◦ The response.text() actually creates another Promise, for handling its data • We can then do whatever we want with the final retrieved data Okay, so it works, but it isn't terribly interesting, right? That's because there hasn't been a benefit yet. We can retrieve more than just plain text. Commonly, we'd want some more involved data. Dynamic dynamos of doom! (or… something) We can load in chunks of the Document tree at a time. There are actually several ways to loads in parts of pages (or even entire pages) at a time, including iframes, etc. Let's try using an XMLHttpRequest, and load in some data. For this example, we actually don't need any PHP (yet). Instead, let's make two HTML snippets: --thing1.html-- COSC1P02 COSC1P03 COSC2P03 COSC2P89 COSC3P71 --thing2.html-- APCO1P50 MATH1P93 MATH1P94 So, how will we use these? Let's make the skeleton of another page (this will eventually be the 'real' page): --dynamish.html--

|  |  |
| --- | --- |
| **Dept** | **Code** |

COSC Other

The idea is we'd like to have a single table, but let it load in different data. Let's make our first attempt at that function, which will definitely (totally) work: function load(page) { req=new XMLHttpRequest(); req.onreadystatechange=handledata; req.open("GET","thing"+page+".html"); req.send(); } Naturally, we'd also need to provide the handledata function, so: function handledata() { if (this.readyState==4 & this.status==200) window.courses.innerHTML=this.response; } And if we reload the page and try it out… Well, darn. What happened? On the bright side: we put the new content into the table perfectly. It integrated into the Document tree just fine. On the down side: we just clobbered the table headings. Crap. Solutions? • We could parse the XML/HTML response, to add each row in • We could serve up the table data in a JSON format, and then set the .responseType of the request to "json", after which we could simply parse the entries and construct the remainder of the table ◦ Of course, that'd mean we'd also need to include code to remove non-heading rows • We could have the served-up table data also include the table headings • We could simply clobber the heading, and then reconstruct it afterwards • We can do what we'll actually do. Let's do that function handledata() { if (this.readyState==4 & this.status==200) window.courses.insertAdjacentHTML("beforeend",this.response); } Give this a try, and see how it works. It works! Right? ...right? Well, technically no. If you try clicking more than one button (or the same twice), you'll see the problem. If you wanted to fix this, you could clear out the existing entries first. Realistically, this is part of why so many webpages are comprised of divs inside divs inside divs inside… Anyhoo, you can tweak it if you like, but otherwise we can continue for the real fun. Dynamic data Before we get to databases, we can't replicate 'real world' dynamic data. However, we can still write something that adapts to arbitrary data, which is all we need. This one is going to use a few files, so you might want to create a subfolder (or not). We'll start with nothing more than three text files (note, in case you don't create a separate folder for all this, these'll need to be the only text files in the folder). --Cakes.txt-- Black Forest,7.0 Christmas Log,6.5 Big Honkin',10.0 Birthday,6.5 Coffee,7.0 --Pies.txt-- Pumpkin,10.0 People,4.5 Purple,8.5 Pumice,1.0 Putrid,0.5 --Monkeys.txt-- Monkey kidneys,1.5 Monkey brains,1.5 Monkey ears,1.5 Monkey nose,1.5 Monkey toes,1.5 Next, we want a PHP script that, when provided with a filename, can build up a table contents: --retrieve.php-- #!/usr/bin/php-cgi

|  |  |
| --- | --- |
| **Type** | **Score** |

|  |
| --- |
|  |

Try it out for now, to see how it looks. By now, everything should be pretty understandable, but it's nice to reinforce the major points, right? • We have a div containing everything ◦ The reason for the inline-block is to make the sizing easier • In the rendered page, we end up with a pattern of spacers and tabs at the top, followed by a line-break, followed by a div that holds a table inside • For the spacers, they're just divs that only have the bottom border defined • For the tabs, for now they have full borders ◦ Taking a look at the selected class, we can see that the bottom border will disappear ▪ (So a tab selected will have left, right, and top borders) • Both spacers and tabs are also inline-block, for simplicity's sake • We get the names for the tabs from a glob on the containing folder ◦ Reminder: a glob is just the name for a pattern that matches on filenames ◦ In this case, anything that ends with .txt ◦ Notice that we're also assigning an id to each, generated from its name/filename • The table tag doesn't have a border set because its enclosing div takes care of that anyway • We actually have two tables, so we won't need to worry about the headings ◦ This is technically foolish, because the column widths won't line up, but it'll look fine for this particular example. Again, a reasonable alternative would be lots more divs • Currently, the selector function hasn't been filled in yet Next, let's work on that function. We already have a gap for where our scripts will go, so we can put it there. Let's start by confirming what we get when we click: function selector(sel) { console.log(sel); } If you reload the page and try it out, there's something to watch out for: the name it receives doesn't match the div's id or the required filename! (Of course, we could easily modify the retrieve.php script to append .txt to whatever it receives. You're welcome to, if you prefer. But I'll just be more careful with the JavaScript instead) We should know enough to be able to populate the entry now: function selector(sel) { req=new XMLHttpRequest(); req.onreadystatechange=function(){ if (this.readyState==4 & this.status==200) { window.entries.innerHTML=this.response; } }; req.open("GET","retrieve.php?selection="+sel+".txt"); req.send(); } We're almost there. If you take a look, it loads the entries just fine. The tabs just aren't… tabby. Here's the final version of the function: function selector(sel) { req=new XMLHttpRequest(); req.onreadystatechange=function(){ if (this.readyState==4 & this.status==200) { window.entries.innerHTML=this.response; prev=document.getElementsByClassName("selected")[0]; if (prev) prev.classList.toggle("selected"); document.getElementById("t\_"+sel).classList.toggle("selected"); } }; req.open("GET","retrieve.php?selection="+sel+".txt"); req.send(); } To clarify, this block here: prev=document.getElementsByClassName("selected")[0]; if (prev) prev.classList.toggle("selected"); is just to 'de-select' the tab the user had been viewing. The conditional is there because, when you first load the page, there isn't a selected tab to deselect. Assuming you've been following along, you should now have a page that can display different content, depending on the contents of a folder. Yay you! By the way, this is all you need for full credit for this lab. It's going to be important for Assignment 3, so it's in your best interest to fully assimilate it. On a side note, it probably would have been better to use POST, instead of GET. If you're interested in more elaborate requests, you can take a look here: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/send#Example\_POST On a side to the side note, we're actually not handling error conditions (e.g. if the data doesn't exist on the server). There's no reasonable way for that to happen here anyway, so it's okay. But it's worth keeping in mind.

**COSC 2P89 – Lab 57 – PHPPython** This week, we'll be focusing on server-side scripting If you happen to have what you made for your previous labs handy, it might help speed one part up a bit. Tasks: For this lab, all I want is a page (or more, if you prefer) with the following: • The first time (and only the first time) you go to the page, display something unique (e.g. legal stuff) • Assemble the page from more than one file • For the full 4 marks (as opposed to 3), have the ability to enter and somehow display some form data ◦ This time, it doesn't need to be for a persistent nickname, though that's acceptable; all that's required is demonstrating that you can accept form data Our first page (anyone else feeling déja déja vu?) We'll start by making a very simple (but still partially-scripted) webpage. We're going to care about permissions, so you'll need a terminal (e.g. PuTTY, or simply booting up under Linux) up for at least part of this. Of course, since we'll be running scripts on sandcastle, unlike with the earlier labs, you'll need to be hosting everything we do on your public\_html. So, create a folder in there (or use the same as before; I'm not the boss of you). In that folder, create a file. I'll call mine simple.cgi. Its contents will be pretty trivial: #!/usr/bin/python print "Content-Type: text/html" print print """

Hello!  
I like cheese!\n How about you?

""" Since we're already repeating, we can repeat the explanations, right?: • The first line is the shebang. Note that, even though it's python, our file ends with .cgi • We used the 'triple-quote' so a string could easily stretch over multiple lines • This time, we're “always coding”, so you can't dump in raw HTML • We'll use print for displaying text ◦ Technically, we can still do fancy replacements, but they're trickier, so I'll skip'em • We can mark strings with " or ' • This time, we need to explicitly include the content type for the HTTP header (and a blank line) Let's try get it running. If you were to go to its address right now in a browser, you'd get a “500 Internal Server Error”1 \*. That's because of how Python/CGI scripts run. The default permissions are 600. The 00 portion is just fine. It simply says no one else is allowed to use the script at all. The 6 means you give yourself (i.e. the 'owner') permission to read, and to write changes to it, but not execute it. Let's change that last part. In the terminal, navigate to that folder, and type: chmod 700 simple.cgi Naturally, if you chose a different filename, use that. 1 Well, I do. Default permissions could be different for you. Try viewing the page in your browser, and it should render correctly now. The explanation for the permissions is the same as for PHP: • Even though 'other people' could view your page, there aren't 'other people' accessing the file, in the sense of other accounts • Rather, the server is becoming you to run the script In your browser, try right-clicking and choose 'View source'. Two things worth noticing: 1. Still no shebang, and still no 'Content-Type…' ◦ That was actually there (to the browser) even with PHP; it just took care of it for you 2. The \n did create a newline, but that still doesn't mean much in HTML So, we're good so far. If you don't want to keep going back to the terminal for permissions, you can just copy the existing script, which should preserve the permissions on the new files. Scripting a page For Python, we're mostly interested in being able to connect to the database (soon!), but we still need to be able to generate and assemble pages. We'll start with the former. For this one, I'll call it dynamic.cgi. This time, we'll jump to the end, and then explain it all together: #!/usr/bin/python import random #Need the 'random' module import cgi, os #For environment data and helpers #Let's get the HTTP header out of the way: print "Content-Type: text/html" print #Let's build the top and bottom of the page first: start="

" end="

" #Note: we aren't printing it yet! #Next, the dynamic bit: randval=random.randrange(1,101) #Exclusive on end addy=os.environ['REMOTE\_ADDR'] msg="Welcome home!" if "139.57" in addy else "New phone; who dis?" msg="Quoth the raven: "+msg page=start+\ "Your lucky number is "+str(randval)+"  
"+ \ msg +\ "  
  
Debugging:" + \ cgi.escape(str(os.environ)) +\ end #Let's see the result! print page Again, set the permissions if you haven't already done so, and let's work out what this all means! • We needed a few modules ◦ random for random numbers ◦ cgi for some helper functions (it'll be extra-useful for forms later) ◦ os is normally for 'Operating System'-related stuff, but this also includes environmental variables (in this case, the user's IP address) • The header needs to be the first thing we print, but doesn't necessarily need to be the first code (though it's safer/easier that way) • Variables don't need $'s. Just type the word you want, assuming it isn't reserved, start with a number, etc. • Technically, it isn't safe to assume the REMOTE\_ADDR key exists in the os.environ dictionary, but… yeah, it kinda is in this case ◦ But you'd encounter a problem if you tried test-running the script without a browser • Even though we know from lecture that tabs/spaces identify blocks, it's safe to indent here because the backslashes let Python know the same expression is continuing onto the next line • Strictly speaking, we don't have a ternary conditional operator, but we can simulate it • That method for checking if you're at Brock isn't 100% reliable. e.g. an address of 100.100.139.57 would say you're at Brock, but this is adequate for our needs Try loading the page. It's okay if it doesn't work at first. Feel free to comment-out pieces, and try each step along the way (obviously make sure to print statements along the way, rather than generating the strings but forgetting to display them). On the other hand, if it doesn't work, maaaybe that's intentional? Because sooner or later, something will go wrong. Let's take a quick break to learn about the CGI traceback module. Debugging Consider the following script: #!/usr/bin/python import random print "Content-Type: text/html" print print "

" print "Random number: "+random.random() print "

" If you try running it, you'll get the “Hi!” title, but nothing else. Why? The script actually fails halfway through, so the browser displays whatever it received before it stopped. Debugging this kind of thing can be very tricky, particularly when you're relying on things like actual browser information, etc. However, we have something to help us: #!/usr/bin/python import random import cgitb #Traceback! cgitb.enable() print "Content-Type: text/html" print print "

" print "Random number: "+random.random() print "

" Ahhh… I tried concatenating a string with a float! (If we cared about making it work, just change the random portion to str(random.random()) ) You might actually want to include those two lines in the rest of your scripts for this lab. Or not. Assembling a webpage It might sound like “assembling a webpage” is what we did before, but this is something else. This time, we're going to look at reusing material. First, we'll need some HTML files: ---top.dealie--- We could navigate here! If you wanted to view this directly in the browser, you'd need different permissions between this and the 700 we've been using. However, it won't matter much what you do, because of how well eventually use it. ---bottom.stuff--- Broccoli University. May contain trace amounts of Cauliflower University ---body.body---

Content unique to *this* page goes here.

---grandstyle.css--- /\*The permissions can matter for this one! Try 644?\*/ body {background-color:grey;} p {background-color:red;} ---subdomainstyle.css--- header {background-color:cyan;} footer {background-color:green;} Okay, so now we have four pieces of a webpage, but no actual page. Let's fix that: ---combined.cgi--- #!/usr/bin/python import random #Need the 'random' module #I left out the cgitb stuff (but you can add it back in, if you want) print "Content-Type: text/html" print #btw, this line tells the browser the header's done #Btw, since we're using a string, be careful # of backslashes. print """

""" #Next, the other pieces! print open('top.dealie','r').read() print open('body.body','r').read() print open('bottom.stuff','r').read() print "

" In this case, combined.cgi is “the webpage”; everything else is just supplementary material. (In a more 'real' example, you might have other folders in play, possibly with different permissions, etc.) So, what's interesting here? • Rather than bothering with a for-each loop this time (which Python also has, as for), we turned the second style file into an embedded stylesheet • This time, I used silly names for most of the components, since the browser never sees them • I still used .css for the external sheet because the browser does see that ◦ And .css for the embedded because… what's that behind you?!? ▪ Moving on… • There are other ways to open files, but because this way is so easy we'll just stick with it • Including Python scripting from additional files is absolutely possible, but honestly unnecessary, and outside the scope of this course C is for bisCuit; that's good enough for me... Let's talk about cookies. As you'll recall, cookies are small snippets of text the server can ask a browser to hang onto (and return in the header of future page requests). They're pretty simple, but potentially very powerful. Since we aren't dealing with databases yet, we won't have much to tie to them, but it's still a good time to get started with the basics. Let's start with those irritating disclaimers lots of websites have now. We could program in some form of agreeing to the terms, etc. But most just go by the, “if you continue, you're agreeing” routine, so we really just need to display a message the first time a user visits, and never again. This kind of thing could be split into a separate Python file, because it keeps the bookkeeping away from the actual content. However, as mentioned earlier, we won't be doing that here. This will be only slightly more complicated than with PHP: • First, we'll confirm that we have cookies (at all) • Next, we'll check if the cookie we want is included ◦ We'll use disclaimer again • If desired, we could then look at its contents paired value (but we don't need that for this) • If this is our first visit, we'll assign the cookie ◦ Unfortunately, we'll be doing it manually • For ha-has, we'll make (and call) a function to do this. Because why not? So, let's take a look, shall we? #!/usr/bin/python import os newbie="Welcome, for the first time!" friendo="Welcome back!" def cooklaimer(): if not os.environ.has\_key('HTTP\_COOKIE'): print "Set-Cookie: disclaimer" #Yes, printing return newbie else: cookies=os.environ['HTTP\_COOKIE'].split(';') for cookie in cookies: if cookie=="disclaimer": return friendo print "Set-Cookie: disclaimer" return newbie print "Content-Type: text/html" msg=cooklaimer() print #Again, finishes the HTTP header print "

" print "hi: "+msg print "

" Again, let's break this down: • We've made a function, and put some conditionals in it ◦ As we covered in lecture, we use indentation to define blocks of code • First, we check if there are no cookies at all for this domain ◦ i.e. not only the first visit to this page, but the first visit to COSC at all • If we have cookies for the domain, we want to see if we have disclaimer: ◦ The cookie string is just one line of text, so we break it up, using the semicolon as a delimiter ◦ The string breaks up into a list of strings, which is iterable ▪ For each string in the list, we check if it exactly equals “disclaimer” ▪ If we had assigned a paired value, we'd need to change this a bit • If we do have the disclaimer cookie, then there's not much to do. We return a string, to be printed at a later time (i.e. after we've actually started printing the document's contents) • Otherwise (whether because we have COSC cookies but not the disclaimer, or we're entirely new), we need to assign the new cookie ◦ Notice that we're printing it here. That's because the Set-Cookie text needs to be part of the HTTP header (because it isn't part of the document contents themselves) ◦ We're doing it simple, so the expiry is session (when you close the browser) ◦ It doesn't actually matter whether this function is invoked before or after the Content-Type line, so long as it's before that extra blank line (that indicates the end of the HTTP header) You could likely find a library or two to handle more of this for you, but we shouldn't need anything fancy. Also worth noting, if you need to escape any text, there are several tools available, including cgi.escape(). With the quirks, we probably won't want to deal with sessions just yet, but we can still handle forms. Let's try that next? Formulaic pages? (bah. insert your own joke here) As mentioned above, we won't be dealing with sessions here. It isn't that we couldn't; they just wouldn't be as easy as with PHP. --Interjection!-- If you do want to do sessions, you have two approaches available to you. One is to find a handydandy middleware. You might have trouble getting it to run on sandcastle, but if you have your own server it should be very easy to set up. Most decent web servers are designed to be extensible, and have 'hooks' for connecting to (e.g. SGI, etc.). Outside of that, baking your own sessions isn't hard. You need two things: persistence, and guaranteed uniqueness of session ids. We'll be dealing with databases soon enough (which is why we're putting this off), but you can use a file (similar to how you're storing data for PHP with your A3). When you create a text file to store all of your records in a tabular format, for easy searching/indexing later, that's a flat-file. If you create a library/application to manage it, you could even call it a flat-file database, but that's a whole other topic. Point is: so long as organization/complexity, concurrency and performance aren't big concerns, you can get persistence pretty easily. For unique ids, there are plenty of tools for fixed-length strings/numbers, including hashing, base64-encoding, and even generating UUIDs. e.g. if you import the uuid module, and call str(uuid.uuid1()), you should get a complex hexadecimal pattern seeded by both the device's host name and current time. We back? Good. Let's get onto the forming! --formic.cgi-- #!/usr/bin/python import cgi import os def claimAnt(): formica=cgi.FieldStorage().getfirst('ant') if formica!=None: open('antipathy','w').write(formica) return formica else: if os.path.isfile('antipathy'): return open('antipathy','r').read() else: return 'radioactive' ant=claimAnt() print "Content-Type: text/html" print print "

"+\ "

Have you ever heard of %s ants?

"%(ant,)+\ "

Top of Form

Whatsyer fav ant?"+\ ""+\ "

Bottom of Form

" Go ahead and give it a try. Note that this is not the same as a session. Yes, data can be stored, but it's stored globally. If you wanted to modify this to behave more like a session, one option would be to check for a session cookie (generating one, if needed), and then creating the datafile with a name derived from the cookie. Of course, you'd likely also want a script to go through every now and then, to remove old files. There isn't much new here: • Most notably, we're using FieldStorage to access what's in the form • This time we're creating a file, in addition to reading it ◦ After testing the script, take a look at the folder: you're the owner of antipathy! ▪ Which makes sense, as the script runs 'as you' • For the sake of the first execution, we're testing if the file exists before opening it ◦ It's written longer than necessary, to fit the geometry of this page better • For actually displaying the chosen ant-type, there's a single example of a string format in Python ◦ (It substitutes the chosen ant for %s) There's just one small problem: when we submit to the form, we end up on that stupid formica.cgi# variant, which also means reloading the page will give the 'do you want to resubmit?' warning. Remember how we handled that with PHP? That's all we'll do here, as well. We just need to tweak the first return of that function. Basically, this: if formica!=None: open('antipathy','w').write(formica) return formica turns into this: if formica!=None: open('antipathy','w').write(formica) print "Location: ./formic.cgi" print exit() And that's it. Since this function is the first thing that runs, it just immediately sends the redirect to the browser. (Feel free to not bother adding this to your own version; it's just good to know it's an option) What else could we do? Well, it's a whole new language, right? There's tons we haven't covered: • Obviously Python is pretty popular for regular scripting as well, and so can be run directly from the command-line. You'd have a hard time testing some of the CGI stuff, though • Note that, in the dynamic.cgi example, we dumped out a listing of the extra information available to the server (including user agent, etc.). You could have some fun with some of that What about the actual submission task? If you went through all of this, all you need to do is copy&paste the cookie example into your formica. Because the form saves the ant type as a file, that technically fits the requirements. Of course, since you're permitted to do this in multiple files, you could also just demonstrate the latter two samples. The point of this lab is for you to get practice, so go with whichever you prefer Addendum: Oddly enough, the line endings problem from PHP is also present here (presumably actually a CGI thing). That means you need to either be careful with your choice of editor, or run the dos2unix command on your script before running it.

**COSC 2P89 – Lab 8 – MongoDB** This week, we'll have the joy of persistence and queryable data! Tasks: For this lab, • You can get 3 marks by simply showing the pies example • For the full 4 marks (as opposed to 3), create a page that lists all of the course codes and titles, but nothing else (e.g. no descriptions) ◦ It can be in a table, unordered list, or even just separate paragraphs or divs ◦ Note that this task is closer to the pies task, than to the provided courses content Getting started: the MongoDB client As with our lecture examples, before we get into the fun stuff, we first need to become at least slightly familiar with the command-line client. To start, PuTTY into sandcastle (or ssh into it, if you're on MacOS/Linux), and type: mongo What we have is basically a very simple JavaScript environment. For example, we can declare variables, perform loops, define functions, etc. (You can even use the arrow-notation for functions, but… why?) If this seems like overkill, it really isn't. Even though systems are normally administered through a customized interface (with appropriate access controls, predefined queries for specific tasks, etc.), getting a system set up can be very different. In particular, batch operations are pretty common. Ostensibly, the mongo client can easily run a JavaScript file for the automated execution of several operations, without the need for direct human involvement (good for either initialization, or scheduled maintenances). Technically, that would be slightly less convenient on sandcastle, as the executable for the client is actually an alias, to make it easier to run. But the ability is still there. In any event, for now, let's just get you into your database, and authenticated. First, we need to tell it which database to use. It's just your username. use username; (e.g. if your username were aa00bb, you'd say use aa00bb ) Next, you need to authenticate. The db object is your entry point into the database collections/operations: db.auth('username','studentnumber'); As you can guess, the password is just your student number. Make sure to send them as strings. e.g. your student number isn't actually a number. Anyhoo, it should return a 1. If you made a mistake, it'll give an error message and return a 0. Assuming we're good to go, we can add some testing data. BTW, if, for some reason, you did need to semi-automate something (e.g. prepping your assignment), you can still use a bit of bash-y goodness. e.g. mongo < mysetupscript You can even //comment, if you like. Adding basic data We'll start with a very simple example: types of pie. (If you haven't already switched to your database and authenticated, do so now) db.pies.insertOne({flavour:'apple',crust:'flaky'}); You'll notice we didn't need to explicitly create the pies collection. Since there's data in it, it now exists. Let's add another: db.pies.insertOne({flavour:'cherry',crust:'sturdy'}); I'd like to add more, but separate inserts seems tedious. Let's use an array: db.pies.insertMany([ {flavour:'steak and kidney',crust:'soggy'}, {flavour:'mathematical',crust:'confusing'}, {flavour:'pie',crust:'crusty'} ]); We'll eventually need more data than this, so let's just add it now: var dbuser='yourusername'; var dbpass='yourstudentnumber'; use yourusername; db.auth(dbuser,dbpass); db.courses.insertMany([ {code:"COSC1P02",title:"Introduction to Computer Science",desc:"Foundations of Computer Science and computer programming in a high-level language (normally Java).",prereqs:[]}, {code: "COSC1P03",title: 'Introduction to Data Structures',desc:"Programming and problem solving in a high-level programming language (normally Java).",prereqs:['COSC1P02']}, {code:"COSC2P03",title:"Advanced Data Structures",desc:"Implementation and use of advanced data structures including trees, graphs, hash tables and advanced list structures, sorting and searching, recursion and traversals.",prereqs:['COSC1P03','MATH1P66','MATH1P67']}, {code:"COSC2P89",title:"Internet Technologies",desc:"Concepts and techniques required for building and maintaining advanced interactive Web sites.",prereqs:['COSC1P03']}, {code:"MATH1P66",title:"Mathematical Reasoning",desc:"Introduction to mathematical INDGraction, logic and proofs including mathematical induction.",prereqs:[]}, {code:"MATH1P67",title:"Mathematics for Computer Science",desc:"Development and analysis of algorithms, complexity of algorithms, recursion solving recurrence relations and relations and functions.",prereqs:['MATH1P66']}, {code:"COSC4P61",title:"Theory of Computation",desc:"Regular languages and finite state machines: deterministic and non-deterministic machines, Kleene's theorem, the pumping lemma, Myhill-Nerode Theorem and decidable questions.",prereqs:['MATH1P67']}, {code:"COSC3P71",title:"Introduction to Artificial Intelligence",desc:"Historic and philosophical foundations of AI; knowledge representation and reasoning; problem spaces; blind and heuristic search; adversarial search in game playing; machine learning; social and ethical considerations.",prereqs:['COSC2P03']}, {code:"COSC3P32",title:"Introduction to Database Systems",desc:"Fundamental database concepts: specification, design and applications; various models including the relational model; normal forms, efficiency considerations, queries using SQL, database administration and security.",prereqs:['COSC2P03']}, ]); Yeah… I don't trust that to copy&paste perfectly, nor do I expect anyone to type that much… So here's what you'll do. Download/save this: http://www.cosc.brocku.ca/~efoxwell/2P89/labs/Lab08courses.txt Important: If you just copy/paste its contents, then make sure the line endings are correct! (using dos2unix) Then, assuming it's in the same folder as you're in (and you're still logged into sandcastle, obviously), edit it to have your username/password/username, and then type: mongo < Lab08courses.txt This is known as a redirect: it uses the contents of the text file in place of standard input (the keyboard). Anyhoo, one way or another, you should probably have the records you need added. Assuming you haven't been having any error messages, you can proceed. (Otherwise, you'll obviously need to take care of those first) On a side note: this really is just JSON. In this case, the prerequisites are stored within an array. You can do something similar for storing related keywords, tags, etc. So, after all of that, do we actually have anything? show collections; Yup. Yup, we do. Let's actually see what's in there! Querying: filters and projections There are two forms of query we'll be using commonly: asking for one thing, or asking for some things. Let's start with the latter: db.pies.find(); That was interesting. Let's say we just want one: db.pies.findOne(); This takes whatever happens to be the first match, and returns it. Obviously, never use it if you're relying on a specific match to be first. You might also notice that it automagically generates a unique identifier for each record. That can be useful, but though we likely won't be needing it any time soon. There are two ways to qualify our queries: filters and projections. Filters control which records are selected. Projections control which parts of each record are returned. Let's start with a filter: db.pies.find({}); Yeah, okay, that one was cheating. This time for real: db.pies.find({crust:"confusing"}); All it does is find records that match the properties listed in the filter. You might be tempted to think that both find and findOne would have the same return here, but that isn't quite the case. Try the following: var c=db.pies.find({crust:"confusing"}); var r=db.pies.findOne({crust:"confusing"}); You'll notice we don't appear to have any results yet. There are hints at what happened, though. The mongo client actually has tab-completion, so start typing r. and press tab twice. Now, start typing c. and press tab twice again. In this case, r is a document, and c is a cursor. You can read some interesting stuff about cursors here: https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/ But the shorthand is that a cursor is effectively like an iterator over the queried data. As such, we have available to us: • c.hasNext() to tell us if there are any more records we haven't seen • c.next() to give us the next record • c.forEach(callback) to call the callback function on each record But that's all overkill for now. If there had been more than one pie with a 'confusing' crust, we would have gotten more record, but they're currently all unique. Let's try finding all pies with crusts ending with 'y' (e.g. 'flaky'): db.pies.find({crust:{$regex:'y$'}}); (We actually don't need to use regex for anything we want; it'll just save us a touch of time sometimes) The $ is the 'end anchor', so this just says: match on anything that ends with a y. Similarly: db.pies.find({crust:{$regex:'^s'}}); yields everything starting with an s. If we wanted everything starting with c and ending with y: db.pies.find({crust:{$regex:'^c.\*y$'}}); In this case, $regex is an example of an operator. We've already seen several other operators in lecture, so refer to those examples for more, and/or read here: https://docs.mongodb.com/manual/reference/operator/query/ and here for a bit more on regex: https://docs.mongodb.com/manual/reference/operator/query/regex/ Okay, so this is all fine and good, but what about projections? As mentioned earlier, it just controls which fields will be included within each document of the results. For example, suppose I was interested in pies starting with s, but only the flavours themselves: db.pies.find({crust:{$regex:'^s'}},{flavour:true}); This almost gives us what we want, but by default \_id is included, so we need to override that. Let's say I'm also only interested in the first instance: db.pies.findOne({crust:{$regex:'^s'}},{flavour:true,\_id:false}); In fact, if we know there'll be a result, we could even say: db.pies.findOne({crust:{$regex:'^s'}},{flavour:true,\_id:false})['flavour']; or db.pies.findOne({crust:{$regex:'^s'}},{flavour:true,\_id:false}).flavour; You get the idea. I think, by now, we're ready to start using these in a webpage. Adding a database to Python We already know how command-line Python works, so we'll jump straight to the cgi. --pies.cgi-- #!/usr/bin/python from pymongo import MongoClient username='yourusername' #Change this! passwd='yourstudentnumber' #Change this! client=MongoClient('mongodb://'+username+':'+passwd+'@127.0.0.1/'+username) db=client[username] print "Content-Type: text/html" print print "

My favourite pie is " print db.pies.find\_one()['flavour'] print "!

" Of course, you need to put your own username and number in there. As before, don't forget to chmod the permissions to 700 before trying it out in your browser. So then, what's the easiest table we could generate from this? For the sake of argument, let's say we aren't interested in a mathematical pie, and we don't care about \_ids. If we were to try something like: db.pies.find({'flavour':{'$ne':'mathematical'}},{'\_id':False,'flavour':True,'crust':True}) it'd certainly work. We'd end up with a pymongo.cursor.Cursor. There are a few things we could do with a cursor. e.g. l=list(somecursor) exhausts the cursor into a list, and then assigns it to l. But honestly we can just traverse it for now. (Note: I'm going to stop including the import, authentication, Content-Type, etc.) print "

|  |  |
| --- | --- |
| " print "  **Flavour** | **Crust**  " c=db.pies.find({'flavour':{'$ne':'mathematical'}},{'\_id':False,'flavour':True,'crust':True}) for record in c: print " |
| "+record['flavour']+" | "+record['crust']+" |