PHP7

Notes open for creative commons use @ developer blog: https://unfoldkyle.com, github: SmilingStallman*, email: kmiskell@protonmail.com*

Learning Resources
Primary: Official Docs, https://laravel.com/docs/6.x
Secondary: https://laracasts.com/

The Request/Response Procedure

-See JavaScript and HTML for more notes for basics of client-server architecture

-Local server can speed up data access by storing it in cache after retrieved from host

-Basic procedure: browser looks up IP of URL → browser sends request to server → listening server looks at request → retrieves page, etc. and sends back to browser

-Dynamic procedure: browser looks up IP of URL → browser sends request to server → listening server looks at request → server retries page, which contains PHP → processes PHP and executes necessary SQL (contained with PHP code)→ retrieve data from executed PHP & SQL queries → server send page to browser (PHP wrapping HTML)

-PHP runs before HTML/CSS returned

PHP, MySQL, JS Basic Interactions

-PHP = scripting language
-PHP general use – modify html, access database, fetch data, etc.

-PHP in .php files, where all PHP code is wrapped inside tag *<?php //code here// ?>* or *<?= // code here ?>* as shorthand
-Everything under <?php ?> is standard HTML code (<html> … </html>) and read by the browser as such

-MySQL is a DBMS, including the DB itself
-PHP can make SQL calls and save results in arrays, etc.

-JS handles the front-end for sites, including asynch updating

-Summary: PHP handles work on server, MySQL manages data, JS & CSS handle presentation

-Ex. HTML displays fields. JS checks to see if field checked or text entered. When user enters info, JS passes to PHP, which resides on webserver. Server checks info based on user input and replies (via PHP & MySQL). Server responds to client and page is updated based on field entries.

Setting Up a Dev Server

-Local server for dev = faster, isolated from prod for testing, security, etc.

-PHP often in LAMP (Linux Apache MySql PHP), WAMP, etc.. If using these stacks, can often install all needed software as a bundle (ex. AMPPS)

-LEMP stack install: https://devanswers.co/installing-nginx-mysql-php-lemp-stack-ubuntu-18-04/

//sudo brew services restart nginx
//sudo brew services list

//sudo nginx
//sudo mysql.server start

Intro to PHP

-PHP typically in *.php* file inside tag *<?php /*code*/ ?>*, but can also include this inside HTML file. Typical to include all PHP first, in tag, then HTML afterwords.

Basic Syntax
//Comment

/ multi-line
comment*/*

-<u>End statements with semi-colon</u>

-<u>All variables must have $ before their names</u> in all instances (deceleration, referencing, etc.)
-ex. *$show_link_options = TRUE;* *if($dealer_key == 'clkdir')*

-<u>No type</u> declared for variables
-Variables can hold values of *"string"*, *$other_var*, int, float, array, object, function def, etc.

Arrays
-Array declaration: *$some_array = array();* *$some_array = [ ];*

-Array access: *$some_array[0]* – zero oriented
-Array initialization: *array("some", "other", "more");* *$some_array = [elm, elm2, elm3];*

-Multi-dimensional array: *array(array(…), array(…), array(…))*
-Multi-dimensional array access: *$some_array[0][0];* *$some_array[ ][ ]= [1, 2, 3];*

-Also, associative arrays, which are similar to a map.
-Ex. *$assoc = ['name' => 'Smith', 'age' => 32, 'city' => 'Chicago']*
$assoc_city = $assoc['city'];* *//$assoc_city === Chicago

Naming
-Variable naming: (start with alpha, etc.), and words separated with underscore
-Function naming: underscore or camelCase word separation
-PHP variable names are case sensitive, function names are not

Operators
-Math: standard math, ++, –, ** (exponentiation)
-Assignment: =, +=, -=, /=, %=, etc.

-Comparators: standard: >=, <=
-equal to (coercion): ==
-equal to (no coercion): ===
-not equal (coercion): *!*= or <>
-not equal (no coercion): *!*==

-Logical: *&&,* ||, *!, xor*
-*!* example: !

-Ternary: *some_condition ? (if true do this) : (else do this);*

-Increment before logic (before test, etc.): *++$some_var*
-Increment after logic: *$some_var++*

Strings
-Literal strings: Single quotes. *'This is a string $x'* //prints *This is a string $x.*
-Template strings: *"This is a string $x"* //prints *This is a string [val of x]*
-Note that you can only use this for variable into string, not function output into string

-Concatenate with . between strings. Ex. *"Output:" . some_function() . " is ouput value"*
-*$first_string .= $concat_string* – concats second to first

-If including PHP as part of string inside embedded html, do:
<a href="somestringhere=<?= $variable_x?>morestring">

-Numeric strings converted to numbers in math, etc. operator statements

String Escape Chars
-If quotes not in pairs put \ before quote

t – tab *n* – newline *r* – return \\ - backslash

Multi-Line Cmds
-Strings can be separated by lines, including blank lines, as long as inside quote pairs

Variable Types
-Very loosely typed and with coercion as standard (ex. *substring($number, 3, 1)*)

Constants
-To define a constant, use: *define("VAR_NAME", value),* which creates a variable of *VAR_NAME*
-Name in uppercase

-When referencing const, do not use *$* before its binding. Also do not need to store to a binding, as PHP separately stores an array to ref all constants. Constants are not variables.

-Constants created as global scope, even if created inside function
-Check if constant has been defined via: *defined("SOME_CONST");*

```
-ex.    define("CONST_ZERO", 0);          //no binding needed
        echo CONST_ZERO;                  //no $ needed
```

Magic Constants
-Commonly used constants that exist in core PHP and are auto populated with data:
-Note, two underscores
-To reference, just *echo __LINE__*

__LINE__ - current line num of file
__FILE__ - full path and filename of file
__DIR__ - directory of file
__FUNCTION__ – function name
__CLASS__ - class name
__METHOD__ - method name
__NAMESPACE__ - current namespace

-If no output (ex. not calling *__FUNCTION__* from function), then empty var
-Useful for debugging

print vs echo
-*echo* simply displays, while *print* returns the value *1*
-syntax: *print "something is printing";*
-*echo* slightly faster since no return

Functions
-Definition:

```
function funcName($var_one, $var_two) {
        //logic;
        return …;
}
```

-Call: *funcName(3, $some_var)*

-PHP functions are first-class and can take in functions as args, store them in arrays, and return them.

-For function that returns function, return inner function as an Anonymous function defined by *function(args){ //body }*
-Must include outer params into scope of inner function by including with *use (args)*

-Syntax:
```
function compose($a, $b){
    return function($c) use ($a, $b){
        return $a + $b + $c;
    }
}

echo compose(1, 4)(2);
```

-Arrays can store anon functions *$fn = ('one', 'two', function($x){ //body });*
-Functions in arrays can return vals

-Can assign functions to variables:
```
    $function = function($a, $b){ //body logic}
    $function(10, "word");
```

-Can also pass in function as arg to function. Ex.

```
  function high($a){            //$a is function name
    $a("dddddd");                       //printMe("dddddd");
  }

  function printMe($a){
    echo ($a);
  }

  echo high(printMe);                   //dddddd
```

Variable Scope
-<u>PHP only has global and function scope</u>, not block scope
-Vars declared within function exist only within function (function scope, local scope)

-<u>Global variables are only available from global scope</u>, and cannot be accessed from within function. Function has an encapsulated scope.
-Ex.

```
          $global_var = "global";

          function echoGlobal(){
                    echo $global_var;
          }
```

```
        echoGlobal();          // will error out as undefined variable
```

-<u>Static</u> variables declared within a function hold a persistent value across function calls to that function.
-Syntax: *static $my_variable = 0;*
-Static declaration statement (usually with initialization) skipped after first call to function
-Ex.

```
   function printVar(){
      static $counter = 0;
      return $counter++;
   }

   printVar(); //0
   printVar(); //1
```

-<u>*global* keyword</u>: To <u>access global variable from within function</u>, set *global $some_global_var;* prior to referencing that var from within the function.
-Beware of using (and global vars in general), as complicates scope vs preference for more specific scope and makes code less maintainable & more likely to have hard to troubleshoot bugs
-*global* Example:

```
        $global_var = 1234;

        function echoGlobal(){
                global $global_var;
                echo $global_var;
        }

        echoGlobal();          //output: 1234
```

-<u>Superglobal variables</u> are similar to magic constants, except allowing access to data passed in from the browser/client
-Accessible from function or global scope

-*$GLOBALS* – array of var names for all global vars
-*$_SERVER* – array with info on server such as headers, paths, etc. Entries determined by server.
-*_$GET*, *$_POST* – array of vars passed to script via http GET or POST methods
-*$_FILES* – array of items upload to script via http POST method
-*$_COOKIE* – array of vars passed to script via http cookies
-*$_SESSION* – array of session vars available to script
-*$_ENV* – array of vars passed to script via the environment method
-*$_REQUEST* – array of info passed from browser (*$_GET, $_POST, $_COOKIE* by default), including key values pairs passed into URL

-Many of items stored in superglobal arrays have keys, which can be used to access their data. Example: *$_SERVER['HTTP_REFERER];*

-ex. *https://myPage.html?dealer_id=1234&style=hd*
    $my_id
-Avoid naming vars in *$_SOMEVAR* format, to avoid confuse with superglobals

-To avoid scenarios where hackers could exploit superglobals, wrap references to superglobals in call *htmlentities($_SOMESUPERG);*

Expressions and Control Flow

Basics
-Expression – simple combo of vals, vars, operators that result in a value. Ex *$y = $x * 2;*
-Statement – combo of expressions that are all part of the same logic flow

-*TRUE* is coerced to 1, *FALSE* is output as no value

-Literal vs variable – literal is a value not stored (ex. *Echo 73;*) while a variable is bound to a name

Operator Precedence
-(multiplication and division) and (subtraction and addition) have the same precedence, processed left to right if in a combo where of equal precedence

-Precedence order: *( ), ++ --, !, * / %, + - ., << >>, < <= >= <>, == != ===, !==, &, ^, |, &&, ||, ? :, assignment, and, xor, or*

-Precedence summary: in/decrement, not, math, comparison > <, comparison ==

-Multiple assignment: $x = $y = $z = 0; //all vars will be 0

Boolean
-*1* and *0* coerce to TRUE and FALSE when tested via logical operators
-Boolean converted to string, converts to 1 or 0

If-elseif-else
```
-if (...) { ... }              //can skip { } if logic is only one expression
 elseif (…) {…}                //note: no space
 else {...}
```

-Can shorten *if ($x == false){...},* etc. via *if (!x){...}*

switch
```
switch (var_or_expression) {
        case “potential_value”:
                //logic
                break;
        case "potential_value_two":
                //logic
                break;
        default:        //if no cases met
                //logic
                break;          //optional if default at bottom
```

Ternary
-*someCheck ? If-check-passes-do-this : if-check-fails-do-this;*

Loops
while (//condition) { …}

```
do {                                    //do will always execute one time before while check
   //do-this-logic
 }
 while (//condition);
```

for($i = 0; $i < 10; i++) { … }

-Can also work with multiple vars via comma separation: *for($i=1, $j=1; $i + $j; $i++, $j--) {…}*
-Can exit from loop via *break*

-Can break from multiple levels (ex. *for* within a *for*) via *break 2;*

-<u>If using loops in global scope, counter variables (ex. *$i)* will exists **outside** of loop, in global scope, as only global and function scope in PHP and no block scope</u>
-Ex: *for(int i = 0; i < 3; i++){…}*
```
     echo $i;          //output: 3
```

-*foreach ($some_array as $index_name){…}* - shorthand for *for* loop, where each index in the array on each iteration is stored in variable *$index_name*

-ex.
```
foreach($person as $name){                //where $person is array of strings
        print $name;
}
```

-To loop through assoc array: *foreach ($some_array as $key => $value){…}*

-ex.
```
foreach($person as $key => $value){
        echo 'the value for $key is $value';
}
```

continue
-Similar to *break* except instead of exiting loop it moves onto the next iteration, skipping any code that occurs below it
-Useful for selectively skipping a part of a loop if a condition is met

Casting & Coercion

-As PHP uses type coercion, if want to explicitly cast output that may be coerced, can do so via: *(int), (double), (float), (real), (bool), (string), (array), (object)* in form:

ex. *(float) $someDouble;* *//double cast to float*

-Can cast as part of expression: *$x = (int) ($a / $b);*

Functions and Objects

-Function names are case insensitive (*somefunction()* could be called as *someFunCTion(),* etc.)

String Functions

-*strrev($some_string)* – reverses string
-*str_repeat($some_string, #)* – repeats string # times

-*strtoupper($some_string)* – string to uppercase
-*strtolower($some_string)* – string to lowercase
-*ucfirst($some_string)* – string with first letter to uppercase, rest in lower

-*explode("deliminator", $some_string)* – separates string into array
    -ex. *explode(" ", "one two three")* *//array[1] of returned is "two"*

Including Files
-To include file into *.php* file: *include("someFile.php");*
-File then exists as if code from that file was pasted where the *include* statement is

-If file *middle.php* includes *bottom*.php and *top.php* includes *bottom.php* and *middle.php*, *bottom.php* will then be included twice, and PHP will error out. To avoid & only include once use:
    include_once("someFile.php"); *//used by default*

Requiring Files
-*require(...)* and *require_once(...)* do the same as *include*, except include lets script continue even if included file doesn't exist. *require* includes and errors out if file doesn't exist.

-Use *require(...)* and *require_once(...)* by default over *include* and *include_once* if file is necessary to script

PHP Version Compatibility
-Get PHP version with *phpversion();* *//*returns in form *5.5.38*

-Determine if PHP core (or any) method exists in current version via:
    function_exists("function_name");
-One way to use: test in conditional statement, where use function if exists, and specify other logic if does not exist

Objects
-Review: Data associated with class = properties. Functions inside classes = methods.

-Review: Interface – methods to access class data, often only a select sum of all class methods to allow for proper encapsulation. Reminder that proper use of encapsulation and abstraction makes more maintainable code as underlying logic can change but as long as old interface methods are the same, code still functions without versioning changes needed.

-PHP sees classes as composites and each new object created from the class as an instance.

Class Basics
-Class declaration:

```
class SomeClass {                          //name class in caps
        public $some_property;
        private $another_prop;

        private function someFunction(){…}

        public function anotherFunction(){...}
}
```

-Can use core function to tell all property keys/values for obj instance via:

```
print_r($some_instance, boolean)
//optional: FALSE (default) bool returns key/value array, TRUE returns string
```

-When setting property values, if setting default value (in root of class, outside of constructor or method) can only set to values, not calls that return values (ex. “hello”, 23) and not function calls that return values. If wish to set values with calls, do inside constructor or class methods.

-*require_once, include*, etc. should be placed in php file <u>before</u> class definition

Object Creation

```
-new ClassName;              //note: no ( )
-new ClassName($argY, argX);            //if class defined with constructor that takes in args
```

Accessing Objects
-Access methods and variables of objects with

```
$instance->some_prop              //access property
$instance→some_method();                //call method
```

-Note that no *$* in front of prop and method names

-Access props declared inside the class from inside the class with *$this->some_prop*

Constructors
-Should be included in function called *__construct()*, which is called when object created:

```
class SomeClass {
        public $some_property;

        function __construct($some){                    //two underscores
            $this->some_property = $some;

            $this->set_props_function();                //calls method defined in class
        }

        function set_props_function(){…}
    }
```

-Note that variables are declared before constructor, then assigned vals inside *construct*. Also note variables inside constructor (and class methods) must reference variables defined inside class using *$this→something;*

Cloning Objects
-Objects bound to variable are stored by reference, not copy:

```
    $objectX = new User();
    $objectY = objectX;             //$objectY references the same instance as $objectX
```

-Clone object with *clone* keyword:

```
    $objectY = clone $objectX;
```

-<u>Can</u> use this for deep clone. If O*uterObject* contains property holding another object (*$innerObject),* can clone and access props, methods of *$innerObject*, etc.

Destructors
-Used to free up resources within an object for manual clean-up, closing DB connection, set state persistence before deletion, etc..

-PHP managed by garbage collector, so destructors auto-called when no more references to object and when script finished. Defining logic within a destructor allows you to execute specific logic during destruction

-syntax: *function __destruct( ) { //logic }*   *//contained within class def*

Adding Properties to Instances
-Can add new properties/values to existing class instance simply by saying:

```
    $existing_instance->new_prop = value;
```

-This is, of course, very much against the standards of OOP, so avoid doing

Class Constants
-Definition: *const SOME_VAR = val;*   *//uppercase*
-Reference: *self::SOME_VAR*

-Must set value during class definition (not during initialization of instance)

Class Scope

-Class variables must be declared as either *public, protected,* or *private*
-Methods can also be declared with specific scope and are *public* by default.

-*public* – available for access outside of class, by other classes, by global in rest of script, etc. Can access via *$some_instance→property_name,* or *$this→property_name* from class definition

-*protected* – available only to methods from class or sub-classes derived from class

-*private* – available only to methods from class

-Example:
public $age = 32;
private function remove_date( ) {…};

Static Methods
-Called on class, not class instance (object)

-Cannot access properties of object

-definition: *static function some_function( ) {…}*
-call: *$some_instance::static_method( );*

-Can get slight speed benefit when using, but only use if absolutely need method that needs to be run without instance existing (ex. *public static void main(String args)* in Java).

Static Properties
-Properties where only one instance of the prop exists across all instances of the class. If change in instance A, instance B will also show change.

-Cannot access with ->, instead, inside class, use *self::$some_static* and outside class use *ClassName::$some_static*

-Useful for counters

-Can also manipulate *static* value outside of class via *::*, etc. but be wary of this as if doing this, often better to just use an instance variable

Inheritance
-Syntax: *class SubClass extends ParentClass { … }*

-In child class, no need to use any *super*, etc. keywords. Can simply call methods and access variables with ->, as if were members of subclass. Same for inside subclass: can access parent class variables, etc. with *$this→some_parent_var.*
-ex. *$some_subclass->someParentMethod( );*

***parent* Keyword**
-Can override by creating method with same name as parent method in child class

-Note: can only have methods of unique names. ie *SomeFunction( ){…}* and *SomeFunction($arg) {…}* is not allowed, despite unique param lists.

-If want to call parent method are overriding from within method you are overriding, use:

```
parent::method_name( )
```

Subclass Constructors
-PHP does not call parent class constructor by default. If need logic from parent constructor to execute to make child class functional, call parent constructor from inside child constructor, and call first:

```
function __construct( ){                    //child constructor
    parent::__construct( );
}
```

final **Keyword**
-Put before function definition if want to prevent inheriting subclass from overriding

PHP Arrays

Numerically Indexed Arrays
-Can add to first empty index (php holds a pointer for this) in array via *$array_name[] = val;*
-Note: if using this, don't need to explicitly create array with *array()*, as array of *$array_name* created during first addition

-Can print array contents with *print_r($arr_name);*

-Can also access via index: *$some_arrary[0]*

array( )
-Used to create an array, where each arg passed into it becomes a value in the array
-ex. *array("one", "two", "three");*

Associative Arrays
-PHP's solution to a map, where each index holds a key and a value

-Can create using *array( )* via:

```
array('keyA' => "valA",
      'keyB' => 123);
```

-Can also add new key/value pairs via *$some_array['keyC'] = val;*

-Can then access vals inside array either by key *$some_array['keyC']* or *$some_array[2]*

-If accessing values in assoc array via key, inside template string, do as:

```
"$assoc[name]"            //no quotes around key
```

foreach...as **Loop**
-Traverses through arrays, specifying *$some_array*, and *$current_index*, where each iteration

holds value of current index

-Numeric: *foreach($that_array as $current_item){…}*

-Associative: *foreach($some_array as $key => $value){…}*
-Will traverse through array and or each key/val pair, two variables (*$key* and *$value)* will be available inside the loop
-Can also traverse accoc using the syntax used for numeric, if only need *$value* access

-If printing array to string, etc., useful to use a *\t* to separate indexes as easy to regex for, etc.

Multidimensional Arrays
-Num Indexed: *array(array(…), array(...), array(…));*
                     $some_array[0][3];

-Associative: *array('key' => array('key' => val,…), 'keyB' => array(…));*
                     $some_arr['outer_key']['inner_key'];

-Associative loop:
```
foreach($outer_array as $out => $o_item)
        foreach($o_item as $in => $in_item) {…}

foreach($outer_array as $o_item)              //key optional
        foreach($o_item as $in => $in_item) {…}
```

Array Functions
-is_array($var_x) - returns boolean or undef if *$var_x* does not exist

-*count($arr_x, optional_mode) - .length* equivalent. Set 1 for mode if want count of outer an inner (multi-D) elm.

-*sort($arr_x, optional_mode)* – sorts array directly (instead of returning sorted) and returns bool for success/fail. Mode = *SORT_NUMERIC* or *SORT_STRING*

-shuffle($arr_x) - shuffles array directly and returns bool for success/fail. On multi-d array, only shuffles outer arrays (indexes inside inner arrays left untouched)

-explode($pattern_x, $string_y) – breaks string at deliminator pattern and stores substrings in array. Pattern not included in array.
    -ex. *explode("...", "This...is...string") == array("This," "is", "string")*

-*implode($pattern_x, $arr_y)* – returns a string from an array, with indexes separated by pattern

-*extract($arr_x)* – imports variables from the array into the php local symbol table. Useful when processing, *GET, POST,* etc.. Useful to put output into local symbol table via *extract.*

-extract($arr_x, EXTR_PREFIX_ALL, 'prefix') – this version names all vars in *$prefix_name* form, where 2nd arg means **_** between name and prefix and name is original name. Use when

array has user controlled keys, to prevent malicious attack and overriting of core keys, etc.

-compact('var_name_a', var_name_b', var_name_c') - takes existing variables (ex. *$var_name_a = 123;)* and creates assoc array, where name is key and val is val. Note, no *$* in args.

-reset($arr_x) – resets pointer kept by array traversal methods to array start. Returns element stored at index before pointer reset.

-end($arr_x) – sets array pointer to end of array. Returns element stored in last index.

Practical PHP

printf( )
-printf("some_string", valA, valB, valC, ...) - Prints to screen, like *echo.* Takes in a string that contains various potential formatting specifiers. The vals passed in fill those spots and format as specified.

-ex. *printf("My name is %s. I'm %d years old, which is %X in hexadecimal", 'Simon', 33, 33);*
//outputs "My name is Simon. I'm 33 years old which is 21 in hexadecimal.

-Start specifier with *%*

-Formatting specifiers:

-*b* – binary int	-*c* – ASCII	-*d* – signed int	-*e* – scientific notation
-*f* – floating pt	-*s* – string	-*u* – unsigned decimal	-*x* – lowercase hex
-*X* – uppercase hex			

-If leave args out, will error out as parse error

-Use example: change RGB colors to hex:
printf("<span style='color:#%X%X%X'>Hello</span>", 65, 127, 245);

Precision Setting
-Also *printf( ),* but with more chars in specifiers to determine precision for display of decimal numbers

-Syntax: .# in specifier, before type. Ex. *printf("Dollar cost: $%.2f, (2,227.13 / 12);*
//output Dollar cost $185.58

-specifier: number following a char – Prints char before number # times specified. Char can be a digit. If just specify number, but no char before, then prints spaces. If char is anything but a digit, precurse with '
-ex. *%#3.2f* prints *###185.58*.

sprintf( )
-Same as *printf( )* but instead of outputting like *echo*, returns formatted string (which can be stored as a var, etc.)

Date/Time Functions
-PHP stores time in standard unix timestamp: the num of seconds since the start o 01/01/1970.

-*time()* - returns num of seconds since 01/01/1970 12am
-To get past or future time, subtract or add *x* num of seconds (ex. *time() + 60 * 60 * 24 * 7 * 4)*

-mktime(hr, min, sec, month, day, year) - args are nums with 0 orientation for time args and 1 orientation for date args. Year range is 1901-2038. Returns seconds for specified date (since 01/01/1970). Dates under 1970 return negative num.

-*date("formatX", timestampY)* – Takes in a string with the desired output format and a timestamp in seconds.
    -ex. *date("l F jS, Y - g:ia", time());*   *//output: "Thursday July 6th, 2017 - 1:38pm"*

Date Constants
-Constants that exist in core PHP and can be passed to *date* functions as *format* arg, etc.

-*DATE_ATOM* – Format for Atom feeds. *"Y-m-d\TH:i:sP"*
    -Ex. "2022-10-22T12:00:00+00:00"

-*DATE_COOKIE* - format for cookies set from a web server or JS. *"l, d-M-y H:i:s T"*
    -Ex. *"Wednesday, 26-Oct-22 12:00:00 UTC"*

-*DATE_RSS* - format for RSS feeds. *"D, d M Y H:i:s O"*
    -Ex. *"Wed, 26 Oct 2022 12:00:00 UTC"*

-*DATE_W3C* – format for W3C. *"Y- m-d\TH:i:sP"*
    -Ex. *"2022-10-26T12:00:00+00:00"*

checkdate( )
-*checkdate(month, day, year)* – if args make a valid date, TRUE, else FALSE.

File Handling
-Can access and modify files on disk where PHP is running. Images, logs, etc..
-Assume file names and paths are case sensitive (linux and unix are)

-All f reading modes (*fread(), fwrite,* etc.) are local read only and not chunked. *stream* methods are chunked and allow remote file reading.

Checking If File Exists
-*file_exists(some_file_location)* – boolean for if file in specified location exists.

Opening & Closing Files

-*fopen("some_file", "mode")* – Opens a file in a given mode. Generally store file ref in var.

-Modes:

-*r* – places pointer at start, reads from file beginning, read only
-*r+* - pointer at start, read from beginning, allow writing
-*w* – pointer at start, write from beginning, write only
-*w+* - pointer at start & truncate to zero length, then write, allow read
-*a* – pointer at end, append to file's end, write only
-*a+* - pointer at end, append to file's end, allow reading

-If file fails to open, *fopen( )* will return FALSE. When open file, should combine *fopen( )* with display of error message, etc. if file fails to open

-Note that pointer stays in updated location after processing files. Ex. Call fwrite(): writes from start until end. Pointer is at end.

-*fclose(some_file+_ref)* - close opened file

Reading from Files
-Once file open can interact with by passing reference var to various functions

-*fgets(some_file_ref)* – file get string. Reads a line from a file. Holds a pointer to current line, so can loop through line by line via repeated calls, including *foreach($some_file as $line)*

-*fread(some_file_ref,* #) - Reads # of lines from file. To read all lines, pass in *filesize(path_to_reading_file)* to second arg.

-Wrap output in <pre> tags to preserve line breaks, whitespace, etc.

Creating a File or Directory
-Create with an *fopen(some_file, "mode")* call, where *some_file* specifies file (and optional location) to create file.

-*mkdir('some_abs_path)* - makes a directory. Must be abs filepath.

-*is_dir('some_test_path)* - returns bool for if arg is directory

Writing to a File
-*fwrite(some_file_ref, "text to write")*

-To write and preserve lines/whitespace, use:

```
$some_name = <<<_END
text-here
   text-here
text-here
_END;                    //this line needs to have no indentation in php file
```

Copying Files
-*copy('original_file', 'new_file')* - Filepaths must be absolute, not relative. Returns *FALSE* if fails to copy. If file already exists, will overwrite.

Moving Files
-*rename('original_file', 'new_file_name')* - Filepaths must be absolute, not relative. Returns *FALSE* if fails to copy.

-If moving to new directory, directory must exist (will not be created if does not exist and move will fail.

Deleting a File/Dir
-*unlink('some_file')* - Deletes file. Returns bool for failed/success. Filepaths must be absolute.

-*rmdir('some_dir')* - Removes directory. Directory must be empty. If errors out, check if dir open with *opendir('some_dir')* and if so, close with *closedir('some_if).*

-*array_map('unlink', array_filter((array) glob("path/to/temp/*")))* - remove all files from dir.

Updating File
-Can update with methods already detailed (*fopen( )*, etc.) and move file pointer to specific position via *fseek( )*
-*fseek(some_file, #a, #b)* - *#b* tells where what line # to move to, and *#a* tells how many chars to move back from once at *#b*
-Second *fseek( )* arg can take *SEEK_SET*, which tells to set pointer at *#a* position and *SEEK_CUR*, which sets pointer *#a* positions from current position (and can take a neg val)

Locking Files for Multiple Acesses
-To allow concurrent access, wrap methods that modify files in *flock( )* method (no need to do so with read only methods)

-*flock(some_file, LOCK_EX)* – sets lock on *some_file.* Returns *TRUE* if file locked.
-Unlock via: *flock(some_file, LOCK_UN*

-ex. *if (flock($fileX LOCK_EX))*
```
    {
        fseek($fh, 0, SEEK_END);          //set pointer to end of file
        fwrite($fileX, $text);
        flock($fileX, LOCK_UN
    }
```

-*flock* does not work on NFS file systems, older FAT, etc., so ensure works during test if putting in code

Uploading Files
-Set attributes for *<form>*: *enctype='multipart/form-data', method='post', action='somewhere'*

-All uploaded files stored in 2D assoc array *$_FILES*.
-Can call *if($_FILES)* to see if any files uploaded

-Can access specific file in *_FILES* via *$_FILES['file-name'],* where *file-name is the name* used by the *<input type='file'>* field that the file was uploaded with
-Can then access individual properties of file by calling above as 2D array: *$_FILES['file-*

name']['name'], etc.

-Once file uploaded (ex. via POST, submitted via form, etc.), can move to new location via *move_uploaded_file(file, newloc).* Use *$_FILES['file']['tmp_name']* for *file* for uploaded if have not already moved file.

$_FILES['file-name']['name'] - name of uploaded file (ex. face.jpg)
$_FILES['file-name']['type'] - filetype (ex. *image/jpeg*)
$_FILES['file-name']['size'] - in bytes
$_FILES['file-name']['tmp_name'] - name of temp (yet moved) uploaded file on server
$_FILES['file-name']['error'] - error code resulting from file upload, if exists

Common Filetypes

application/pdf	image/gif	multipart/form-data	text/xml
application/zip	image/jpeg	text/css	audio/mpeg
image/png	text/html	audio/x-wav	image/tiff
text/plain	video/mpeg	video/mp4	video/quicktime

Form Validation
-Use to avoid maliciously formed input data. Typical check = limited filetypes, standardized prog generated filenames, etc.
-If keeping user filename, allow only alphanumeric chars, often lowercase only

System Calls
-*exec(cmd, output, status)* – Can execute sys commands with (*ls, mkdir, grep,* etc.). Only *cmd* arg required.

-Good form to wrap cmd via *escapeshellcmd(cmd)* as arg for *exec,* as santizes cmd of potentially malicious random special chars, etc., and helps prevent hacks

-When output spans multiple lines, returns an array, with one line per array (ex. echo all lines of *ls -la* cmd by iterating through with *foreach*)

Intro to MySQL & Mastering MySQL

-For notes from chapters 8 and 9 of *"Learning PHP, MYSQL, and JavaScript – With jQuery, CSS, and HTML5"* (2018), see *MySQL.odt* document, which covers review of SQL (queries, table/DB creation/modification/deletion, etc.), as well as DB design, normalization, etc.

Accessing MySQL Using PHP

Overview
-Connect to DB. Perform query using string *SELECT*, etc.. Retrieve and format results. Display in HTML, JSX, etc. Repeat for all needed data. Disconnect from Mysql.

Login File

-To quick login across multiple scripts, create a *login.php*, etc. file with login details, to be included by various scripts:

```
// login.php
<?php
	$host = 'localhost-or-remote-ip';
	$db = ‘dbName';
	$user = 'username';
	$pw = 'password';
?>
```

Connecting to DB
-Included login file in via require_once: *require_once ‘login.php’;*

-Connection method:		*$conn = new mysqli($host, $user, $pass, $db);*
-Returns a mysqli object, which is a mysql “extension” built with php for handling db drivers, connections, api, etc.
-If connection fails, returns *false* so can check by checking for *!$conn*

-ex.

```
	<?php
		require_once 'login.php';
		$conn = new mysqli($hn, $un, $pw, $db);
		if (!$) s
			die("Fatal Error");
	?>
```

-Prod version of *die* would be something like, non-program killing functions, that prints:

```
function mysql_fatal_error(){
		echo <<< _END
	We are sorry, but it was not possible to complete
	the requested task. The error message we got was:

	<p>Fatal Error</p>

	Please click the back button on your browser
	and try again. If you are still having problems,
	please <a href="mailto:admin@server.com">email
	our administrator</a>. Thank you.
	_END;
}
```

Querying

```
	<?php
		$query  = "SELECT * FROM classics";
		$result = $conn->query($query);
```

```
            if (!$result)                       //returns false if no result
                    die("Fatal Error");
        ?>
```

-*query()*
- -performs query on DB
- -returns *false* if failed query
- -success, returns *mysqli_result* instance, or *true*, if not pull query

-*mysqli_result* - contains properties of query (ex. *num_rows*) and has methods to pull data from results (ex. *fetch_row())*

Accessing Fields
$result→data_seek(int)
-Adjusts the result pointer to an arbitrary int row in the result
-Traversable via *foreach*

Fetch Single Row
$result→fetch_assoc()
-Fetch a result row as an associative array, without moving pointer, null if no row
-ex. *echo 'author name: $result→fetch_assoc(2)['author']';*

Fetch All Rows
-*$result->fetch_array(TYPE)* returns a row and moves pointer to next row, returning null when no rows left. Can fetch rows using:

```
  while( $row = $res->fetch_array(MYSQLI_ASSOC))
    print_r($row);
```

-*fetch_array(TYPE)* takes three types:
- -*MYSQL_ASSOC* – 2D array of inner assoc arrays
- -*MYSQLI_NUM* – 2D array of inner indexed arrays
- -*MYSQLI_BOTH* - weird monstrosity 1D array

-Can also call: *$result→fetch_all()* which returns an an array of all rows, of row type either *MYSQLI_ASSOC, MYSQLI_NUM, MYSQLI_BOTH*

   -$*result->fetch_all()[3];*   //values from row #3 (zero indexed)

-Above useful when combined with *array_map(), array_reduce(), etc*. calls, *foreach()* traversals, etc.

Basic XSS Atack Protection
-Wrap all fetches in *htmlspecialchars()* call

Closing a Connection
-Small stuff, connection closed when script ends auto, so no worries. Larger, close.

-$conn->close();

-*$result->close();*

Superglobal Review
-*$_SERVER* – array with info on server such as headers, paths, etc. Entries determined by server.
-_*$GET*, *$_POST* – array of vars passed to script via http GET or POST methods
-*$_FILES* – array of items upload to script via http POST method
-*$_COOKIE* – array of vars passed to script via http cookies
-*$_SESSION* – array of session vars available to script
-*$_ENV* – array of vars passed to script via the environment method
-*$_REQUEST* – array of info passed from browser (*$_GET, $_POST, $_COOKIE* by default), including key values pairs passed into URL

***$_POST* Array**
-Browser sends user input to PHP via either *$_GET*, *$_POST*, or *$_REQUEST,* declaring type in *method* for form, etc.
-Associative array holding key value pairs passed in via HTTP *POST* method

-mysqli object (*$conn*, etc.) has function *real_escape_string($someString)* which escapes special chars from string. Use to clean input before query.

-Syntax: *form<action='myphpfile.php' method='post'>*

-forms creating in HTML nested in PHP can call *post* method, which passes form data into *$_POST,* in key→value pair with keys based on html *name* attr, and value on <*input*>, etc. state.

-Prefer over *GET* as less easy for hackers to take advantage of, and *GET* results in larger server logs and bigger urls

***$_GET* Array**
-Assoc array of key=value pairs from URL (ex. *mySite.com?id=1234&size=large)*

***$_REQUEST* Array**
-Assoc array containing contents of *$_GET, $_POST,* and *$_COOKIE*
-Changing *$_REQUEST* does not included arrays and vice versa

-Warn against using *REQUEST* as fact it contains cookie means leaves site more open to cookie based injection attack

Example – Basic Insert
1) Require login file. Connect. Handle error if connection fail.
2) Build HTML form inside *<?php*
3) on-click of button, get info using *$_POST* and validate/clean
4) Create *Insert* query using *POST* variables
5) Run query via *$result = $conn→query('Insert…..');*
6) *if(!$result) { //error handing }*

-If inserting into *auto increment* DB field, pass in *NULL*

Example – Basic Delete
1) Require login file. Connect. Handle error if connection fail.
2) Build HTML form inside *<?php*
3) Onclick of button, get info using *$_POST* and validate/clean
5) Run query: *$result = $conn→query('DELETE…WHERE post_var='something");*
6) *if(!$result) { //error handing }*

-If want to delete without *delete* button click, etc., pass in *delete->true* or such, or just check for some other condition and build *WHERE* based on that

HTML in PHP
-Suggest staying in *<?php* always instead of dropping out to *<html>* document

```
<?php
echo <<<_END
        <html here>
_END;
?>
```

CREATE TABLE
-Should be careful about users explicitly creating a table as could use to damage DB and end up w/ countless tables

-Create using standard *CREATE* statement passing into *query()* on DB connection

Create DB
-First create connection via *mysqli_connect* object passing in usual *host, user* info
-Then, create via standard "CREATE DATABASE myDB"

DESCRIBE Query
-Returns 2D array where first row is DESCRIBE column header ['*Column', 'Type', 'Null', 'Key']*, then each following row is description of one field ['*id', 'smallint(6)', 'no', 'PRI']*

Dropping Table
-Again, should not be available to most users

-Standard *$connection->query('DROP...');*

Displaying Data & Architecture
-Use single responsibility principle.

-Best is to have back-end PHP API, that is fully separate from front end. All front end does is run queries. Front end PHP via url (localhost if some host, etc.) and passes data via *POST* %attribute, then PHP has access to via post.

-If not doing API, try and have front end and back end separated as much as possible. Create a file to create a run queries. Another to build tables. Etc. DRY. Single Reposonsibility.

Hacker Prevention
-By passing in an unpaired quote, etc., a hacker can use this to modify a query, which is just a string (ex. Drop off end of query string and add own end). This is a *SQL injection*.

-To avoid, always run data inserted into query through *$conn→real_escape_string($my_data)* which is a *mysqli* method to remove

Prepared Statements
https://www.php.net/manual/en/mysqli.quickstart.prepared-statements.php

-High efficiency statement executed in two parts: prepare & execute

1) Prepare – statement template sent to DB server. Server checks syntax, then initializes server resources for later use.

-ex. *$stmt = $conn→prepare('INSERT INTO myTable VALUES(?,?,?,?)');*

-*?* act as placeholders for data to be later inserted into query template

-prepare() - sends to DB and returns statement to be used for calling prepared statement or *false* if error

2) Execute – pass in args to *statement* via *bind_param(args)* call, then call *execute* on it, which runs statement

-ex. *$stmt->bind_params('1234', 'Tom', $someArg, $argX);*
$stmt→execute();

-Check statement ran by accessing *affected_rows* prop of *statement*, which holds num of rows changed, inserted, or deleted by last *execute()*

-Once done with statement, free DB resources by calling *$stmt->close()*

Placeholders & Security
-As prepared statements are sent to DB before values for statement, prepare statements are not able to be modified by *sql injection* attacks. Using prepared statements thus provides pretty sure-fire protection.

Preventing HTML Injection
-*Cross-site scripting*, aka an XSS *attack*

-When have a page that allows user to input data into website (ex. a comment form), then user uses this to insert html (or more often JS) into page, which can steal cookies, insert a trojan, etc.

-Can prevent by calling PHP global method *htmlentities($string)* on, which then converts to string, with special chars converted to *html* codes (*&#41;* etc.)

-Useful to create function that first calls *$conn->real_escape_string($input)* on input first, then calls *htmlentities()* on that

-ex.

```
  function mysql_entities_fix_string($conn, $string)
  {
    return htmlentities(mysql_fix_string($conn, $string));
  }

  function mysql_fix_string($conn, $string)
  {
    if (get_magic_quotes_gpc()) $string = stripslashes($string);
    return $conn->real_escape_string($string);
  }
```

Procedural MySQLi
-Can also create call *mysqli* procedurally, as follows:

```
$link = mysqli_connect($host, user, $pass, $db);
if( mysqli_connect_error() ) die(“Connection error”);

$result = mysqli_query($link, “sql-statement-here”);

$num_rows = mysqli_num_rows($result);
$single_row = mysqli_fetch_array($result, array-type-here);

mysqli_insert_id($result);            //return insert id of insert that returned $result

$escaped = mysqli_real_escape_string($link, $val);

$stmt = mysqli_prepare($link, 'INSERT INTO classics VALUES(?,?,?,?,?)');
mysqli_stmt_execute($stmt);

mysqli_stmt_close($stmt);
mysqli_close($link);
```

Form Handling

-Forms: the primary method for a user to interact with DB data

Basic Flow
-Form built in html with *method* for what to submit and *action* for where to submit. User enters *input* data. On submit, passed by JS to PHP. PHP reads data, typically does error checking on, then submits to DB and displays submission confirm message. If error in submission, directs user to correct and does not submit.

-ex.

```
<form method="post" action="formtest.php">
	What is your name?
	<input type="text" name="name">
	<input type="submit">
</form>
```

Retrieving Form Data in PHP
-PHP file specified in *action* then has access to form via *method* submitted to it (*$_POST,* etc.), where *name* of *<input name='something'>* is referenced via *$_POST['something']*, etc.

-Use *isset($_POST['someinput'])* to check if data submitted or <*input*> left blank

-Can build semi-reactive page by passing *post*, etc. to self (form defined in same php file data passed to), then check for *$_POST*, etc. data on page load, and use to modify how page renders. Then, when submit form, page will reload, but now with *$_POST* data from form available to it.

Default Values
-*value* attribute in <*input*> can be specified to submit default value if no value specified by user.

<input type='text' name='percent' value='100'>

-*value* will also show as default value (similar to placeholder) on form.

***text* Input**

Attributes:
-*maxlength='length'* - specify max char length for input
-*minlength='length'* - specify min char length for input
-size=*'length'* - specify width of *input* by num of chars (for font size of *input)*

***textarea* Input**
-larger than *text* as can be many rows longs

Attributes
-*cols = '12'* - optional. Sizing.
-*rows = '33'*- optional. Sizing.
-*wrap = 'type*' - types: *hard*, *soft*, *off.* Optional.
-*maxlength='length'* - specify max char length for input
-*minlength='length' - specify min char length for input*

Checkboxes
-*<input type='checkbox' name='someName' value='someVal' checked='checked'>*

Attributes
-*checked = 'checked'* - include to have checked by default
-*value = 'someVal'* - is optional. By default submitted value if checked is string *'on'.* Use to submit custom val if checkbox checked instead (ex. *1*)

-Can have grouped checkboxes by giving all checkboxes same *name* with different *value*
-If multi checkboxes w/ same string name, only last checkbox val will submit.

-To allow multi-values for multi checks to submit, turn name into name='*someName[]'* for all objects:

```
  Vanilla <input type="checkbox" name="ice[]" value="Vanilla">
  Chocolate <input type="checkbox" name="ice[]" value="Chocolate">

  $ice = $_POST['ice'];              //array, where $ice[0] == 'Vanilla', $ice[1] == 'Chocolate'
```

Radios
-*<input type="radio" name="time" value="someVal" checked="checked">*

-Only one allowed to be selected at a time. Buttons should be group together by all having same *name*. *value* of selected button will submit.

Html attributes
-*checked='checked'* - optional. Should only be on one radio, which will be checked.

Hidden Fields
<input type="hidden" name="someName" value="someVal">

-Submit to form, but not displayed on page (ex. value that changes after form submits, hidden id, etc.).

Select

```
<select name="name" multiple="multiple">
   <option value="valA" >optionA</option>
   <option value="valB" selected="selected">optionB</option>
</select>
```

-Each <*option*> should have a different *value*. Selected one is submitted to *name*

html attributes
-*multiple = 'multiple'* - if added, can select multiple, *value* is an indexed array accessed *name*
-*selected = 'selected'* - value for <*option*> to set as default selected

Labels
-For proper semantic html, labels for form items should be contained within <*label*> tag

```
   <label>
        Username:
        <input type="text" name="username">
```

</label>

Submit Button
<input type="submit" value="buttonText" src='someImage'>

html attributes
-*value* – holds text for button
-*src* – should point to an image, which will used as graphic for button

Sanitizing Input
-Use html attributes and specific <*input*> types as much as possible to limit size, input type, etc.

-In addition, sanitize strings and sql with PHP functions

I) *stripslashes($string)* – removes slashes from input
II) *htmlentities($string)* – changes html tags from input and into *&lt;, b&gt;, etc.*
III) *strip_tags($string)* – removes angle brackets entirely

```
function sanitizeString($string){
        return stripslashes( htmlentities( strip_tags($string) ) )
}

function sanitizeSQL($string, $conn){
        return sanitizeString( $conn->real_escape_string($string) );
                                        //escapes special chars
}
```

HTML5 Enhancements
-*autocomplete* – Attribute can take *'on'* or *'off'*. If *'on'* <*input*>, etc. will remember past user input. Can use with <*input*> of *type: color, date, email, password, range, search, tel, text,* or *url*.

-*autofocus='autofocus'* – Attribute. HTML element will be focused on when page loads. Any elm type.

-*placeholder='text'* - Attribute. Shows light grey default text to be show in <*input*>, which vanishes when user enters text.

-*required='required'* - Attribute. Any <*input*> type. If set and user does not enter input, error will show and form will not submit until input entered.

-*height='someHeight'* and *width='someWidth'* - Attributes. Allow for sizing via html instead of CSS.

-*min* and *max* – Attributes, for inputs *number, time,* etc. Allow range to be set where nums, etc. outside of range are not allowed.

-*step='someNum'* - for <*input*> with up/down arrows in view to switch between nums, etc. Sets step between nums. If *step* == 2, clicking num up will result in 2, 4, 6, 8, 10, etc.

-*form='formName'* - Attribute. For <*input*>*, <select*>, etc. elements that exist outside of <*form></form*>*,* to denote them as part of the specified *formName* <*form*>

-<*input type='color' name='color'*> - will show color picker when picked for user to pick color from

-<*input type='number' name='someName'*> - Only allows nums to be entered and shows up and down arrows to move up or down a step. Useful with *min, max, step* attributes.

-<*input type='range' name='someName' min='0' max='100' value='50' step='1'*> - Shows a drag-able slider, where value submitted is based on position slider moved to.

-<*input type='time' name='someName' value='12:34'*> - Formats input to date, time input and when click on *input box, calendar*/time selector will open. Can also use types *date, month, week, time, datetime, datetime-local.*

jQuery Ajax & Fetch

AJAX 101
-Asynchronous JavaScript and XML

-Major browsers implement *XMLHttpRequest* object, which allows object holding data to be sent on page without page reload

-JavaScript library AJAX uses *XMLHttpRequest API* to send such data. JavaScript sends request to specified url (ex. api url, php file), which gets data sent by Ajax, then sends back response.

-jQuery provides standard library of Ajax *get*, *post*, etc. methods for simplified Ajax use.
-jQuery Ajax can also send/receive plain html and json, which is more commonly done than xml

-GET - Use when only getting data from server, but not modifying it.
-POST – use when changing data on the server.

-Available data types:
  -*text* – transporting strings
  -*html* – transporting blocks of html
  -*script* – adding a new script to the page
  -*json* – transporting json formatted data
  -*jsonp* – transporting json formatted data to external domain
  -*xml* – transporting *xml* data

-Ajax is asynchronous, Ajax functions thus also must take a callback, which gets the *response* as the arg for handling *response* when asynchronous call finishes:

```
$.get( "foo.php", function( response ) {
    console.log( response );
});
```

Ajax Methods

-95% of the time only need to use $.*ajax()* method.

-Syntax:

```
$.ajax({
           url: ‘myPath.php’,
           data: myData,              //ex. { id: 23 }
           type: ‘httpMethod’,        //ex. POST
          dataType: ‘dataType’        //ex. json
         //additional options here
}).done( function( response ){
         //runs if successful response
}).fail( function( xhr, status, errorThrown ){
        //runs if call fails
        //above attributes available to handling logic
}).always( function( xhr, status ){
        //runs regardless of fail or success
});


$.ajax({
    type: ‘POST’,
    url: ‘myurl.com/api’
    data: $(‘#my_form’).serialize(),
    dataType: ‘json’,
    success: function(data){
         //success logic
    },
   error: function(response){
         //failure logic
    },
    always: function(response){
         //always logic
    }
});
```

-GET: *$.get(location, optional-data, callback-on-success(data), optional-data-type-returned’)*
-ex.   *$.get("result.php", { name: "Zara" }, data => $('#stage').html(data), json);*

-GET: *$.post(location, opt-data, callback-on-success(data), opt-data-type-returned’)*
-ex.   *$.post("updateCustomer.php", { name: "Neil" }, data => $('#stage').html(data), json);*

-optional data type returned: *‘xml’,’html’, ‘script’, ‘json’, ‘jsonp’, ‘text’*

-LOAD: *$(‘selector’).load(location, optional-data, optional-callback(data))*
-similiar to GET except, loads data returned directly into *selector* element(s)
-Useful for filling in text elements with simple strings, etc.

-GETJSON: *$.getJSON(location, optional-data, optional-callback(data))*
-Gets JSON data from server location, then passes into *callback*, then can access JSON via *data.propName* inside *callback*
-Useful for filling out text elements with multiple JSON props

-GETSCRIPT: *$.getScript(‘location.js’, optional-callback)*
-Gets javascript, loads, and executes
Ajax Options
-Can pass in additional options to ajax besides required *url*, *type*, etc.

-*async: false* – AJAX runs without asynch calling

-*crossDomain: true* – forces cross domain call. For use with *jsonp*, etc.
-*dataType: ‘type’* - expected response type from server. x*ml, json, script, html.*

-*headers: {//object}* – holds key/value header pairs sent with request
-*password: ‘someString’* - sends pass to be used in auth during call
-*username: ‘someString* – sends username to be used in auth during call

-*always: someFunction(response){}* - runs when ajax completes
-success*: someFunction(response){}* - runs if ajax request fails
-*error: someFunction(response){}* - runs if ajax request fails
-*timeout: someNum* – how long to wait in ms before considering request failed

Ajax & Forms
-*serialize()* - jQuery method can be called on a form (ex. *$('#edit_vdp_goal_form').serialize())* to convert form inputs into query string

-*serializeArray()* - jQuery method can be called on form to convert inputs to array of objects, each object with *name* and *value* key-value pairs

-For client-side JS validation, call event handler *submit* on form and check validation logic only calling *$.ajax(….)* if validation passes

```
$( "#form" ).submit( function( event ) {
     if( //validation condition ) { event.preventDefault() }
     else { $.ajax(….) }
}
```

Fetch
-Successor of AJAX

-JS global interface for making HTTP requests. Uses promises.

-Calling *fetch* returns a promise, that resolves to a *response* obj that holds info on server's response, which can be retrieved via *then* method, which takes a function as an arg to process response

-ex. *fetch("example/data.txt)".then(response => console.log(response.status));*

-Syntax: *fetch("urlRequesting")* - If no protocol name *(*ex, *http:*) in *fetch*, treats url as relative (pointing to folder page is hosted on)

-response.headers.get("key-name") - returns map like object where keys are headers

-*fetch* has a *.text()* method, which is called on the *response* object and returns a promise that is resolved to content of response via *text*. Access via *.then(handlingFunction)* method called on *.text()*.

-ex. *fetch("example/data.txt").then(response => response.text()).then(text =>*

console.log(text));

-*fetch* also has a *.json()* method, that acts the same as *.text()*, but whose promise resolves to json content (or is rejected if not json). If want to convert json to string, call *JSON.stringify(someJson)* on
-Can use *json()* to fetch rest API data

-Can specify methods via *fetch* via *fetch("urlHere", {method: "DELETE"})*
-Can set headers via *fetch("urlHere", {headers: {Content-Length: "65585"} });* Some headers (ex. *Host*) set by browser by default.

Fetch + PHP Example

```
fetch(‘myfile.php’,
        {method: ‘get’}
       ).then(function(response) {
             if(response'.status != 200 ){
                    return response.text();
             }
            else{
                    //do these things
           })
);
```

PHP Return Json
1) Set header at top for json
    header('Content-type: application/json');
2) Echo in json form
 echo json_encode($array)

Form Data & Fetch
-To send form data, inside form submission handling function:
 let formData = new FormData(event.target);

-Then in addition to sending *method* with *fetch* also send:
body: formData

-ex.
{ method: ‘POST’, body: formData }

-Then can access in PHP via *$_POST*, etc.

Cookies, Sessions, and Authentication

Cookies Intro
-File site saves on client computer, up to 4kb, cross-session
-Often used for session tracking, maintaining data across multi visits, shopping cart contents, login details, etc.

-Can only be read by the issuing domain (ex. myPage.com cookie cannot be read from theirPage.org)

-Third party cookies – issued by parts of a site from domains other than the main site domain (ex. *myPage.com* hosted on serverX but component in page hosted on serverY and issues cookie). Often advertising cookies.

-Transferred with headers, before html sent. Client requests page. Cookies and headers declaring type of content being sent, etc. sent. Client receives. Web page contents sent. Client receives.

-Once client has cookie, all further requests to that domain will also send cookie back to site. Cookie only sent to client once, though.

Setting a Cookie
setcookie(name, value, expire, path, domain, secure, httponly);

-PHP global function

-As cookies sent with headers, must call *setcookie(…)* before any *HTML* has loaded

Args
-*name* – name of cookie for future server reference. Alphanumeric.
-*value* – contents of cookie. Alphanumeric. Up to 4kb.
-*expire* – optional. Time expires. Often *time()* + *numSeconds* (ex. *time() +259200)*
-*path* – optional. Path of cookie on server. Where cookie is set for site. If set to subdomain, only used for subdomain. If set for */* is set across whole site.
-*domain* – optional domain to use cookie for. ex. *webserver.com, images.webserver.com*
-*secure* – optional. Boolean. If *TRUE* cookie must be sent over SSL (*https)*.
-*httponly* – optional. Boolean. If *TRUE* must be sent using http protocol (JS cannot access).

-ex. *setcookie('location', 'USA', time() + 60 * 60 * 24 * 7, '/');*

-To prevent need-less re-sending of cookie after initial send, enclose base *setcookie in* in
if(!isset($_COOKIE['my_cookie']){….}

Accessing a Cookie
-Access via *$_COOKIE['cookieName']* super-global, which returns contents

-*ex. $_COOKIE['location']*

-In order to access cookie, cookie must exist, meaning client must access page first (which then sends cookie to server with page request)

Destroying a Cookie
-Is no actual destruction method in PHP, so instead call *setcookie()* again with all the same arguments, except with a *time()* set in the past.

-ex. *setcookie('location', 'USA', time() - c, '/');*

-Can also pass in *FALSE* instead of past time, but useful to use past time, in case client clock not set properly

Cookie Use
-When request page, page sends cookie to user. On next visit, as long as cookies not cleared, cookie still exists. Thus, can then access cookie and read variables from it.

-To 'update' cookie, must call *setcookie(…*) with updated args and cannot update via *$_COOKIE*. This sends new cookie to user, which will be loaded on next refresh.

-Update *setcookie(...)* is a result of cookie flow: User requests page. Page sends headers and cookies. User confirms receive. Page sends HTML. On refresh, user sends cookie again. Cookie itself is read only, though. Thus, PHP must read existing cookie, then update data read, then set to new cookie. Then, on page_reload, user will download new cookie, with updated data.

-ex.
1) if no cookie, *setcookie* with counter = 0
2) if cookie, ++ counter
3) *setcookie* again with updated data
4) on page reload, user downloads new cookie with updated data

-If want to store array in cookie, must *encode* and *decode* to to *json*. Ex:

```
setcookie('userInfo', json_encode(['userID' => 123, 'location' = 'USA');
json_decode($_COOKIE['userInfo'], true);            //pass in true if assoc
```

If 502 Error
-If 502 error from too large upstream add following in php block in nginx *sites-available*

```
fastcgi_buffers 16 16k;
```

    fastcgi_buffer_size 32k;

HTTP Authentication
-Basic website auth

-Flow: PHP sends header to user to start auth request. Auth login, from browser then pops up asking for user/password. User enters value. Page then reloads, now with auth variables accessible in global PHP variables.

-*$_SERVER['PHP_AUTH_USER']* and *$_SERVER['PHP_AUTH_PW']* hold auth input
-Should call *htmlspecialchars(…)* on before processing input to prevent xss attacks

-Basic auth involves to parts:
```
    header('WWW-Authenticate: Basic realm="Restricted Area"');    //Msg added to auth popup
    header('HTTP/1.0 401 Unauthorized');                  //header to display auth popup
```

-Basic auth could be an *if-else* with logic
```
   if(auth details isset){ check details and continue If good}
   else { //headers follows by die("Must login msg") }
```

-Above logic checks for *$_SERVER* variables to see if exist, and if do and correct, continues to load page. If variables not set, then sends *401* and message headers, which causes browser to show login info. After login, page loads again. This time, login details exist in *$_SERVER* variables, and page checks, then continues loading. If user clicks cancel on login, page does not reload and logic continues to *die(…)*, to display error.

Storing Usernames and Passwords
-To store passwords, first convert to a standard one way hashing function use php global function *password_hash(…),* where same password input always returns same hash.

-Syntax: *echo password_hash("somepassword", PASSWORD_DEFAULT);   //return hash*
                                                                //says to use most secure hashing algo
-As same pass in, same pass out, store hash in DB instead off pass, then check pass by running through *password_hash(…)* and comparing hash in DB to hash from user pass input to make sure same.

-*password_hash(…)* uses salt by default to increase hash security

-password_hash(…*)* can take in third parameter via assoc array including options. One option is *['cost' => 10]* where number determines amount of processor time to allocate for hashing. 8 to 10 is good baseline for this, more if fast webserver.

-Set DB to allow 255 chars for hash column, to allow for long hashes

-To check password and compare to hash, use *password_verify($pass, $hash)*, which returns *TRUE* if *$pass* hashing results in *$hash* or *FALSE* if not match.

Setting Cookies in PHP Called from Fetch
-When tried to set cookie in PHP called by JS fetch, cookie not set properly

-Cookies set before HTML sent, so if called from page where HTML has loaded before *fetch,* set the cookies with javascript instead

-*ex*.

```
  let date = new Date();
  date = date.setTime(date.getTime() + 12 * 60 * 60 * 1000);          //12H

  let user = $('input[name='username'dd).val()
  let pass = $('input[name='pass').val();

  let cookieJson = JSON.stringify({
     'user': `${user}`,
     'pass': ${pass}`
  });

  document.cookie = `login=${cookieJson}; expires=${date}; path=/;`;
```

Intro to Sessions
-A server-side store of info (cookies = client-side) that exists throughout a user's interaction with a site, where a unique ID for session is stored in client-side cookie

-Every time client makes an *http* request to webserver, session id passed to server

-Session exists until user closes browser

Sessions vs Cookies
-Cookies set data server side via PHP, then passed to client, then client passes back to server on each page load. PHP then has access to cookie data for use in back-end, before html loads

-Sessions set data serve side via PHP, then pass session id cookie to client, then client passes back session cookie on each page load. Using session ID, PHP then has access to previously set *$_SESSION* variables, whose value is a stored in a client side temp folder set by PHP.

-Sessions are generally preferred over cookies, as session data is stored server side. Client side data via cookies is higher security risk, as hackers could MITM cookies. Since sessions only exist until browser closed, if needed multi-session length store, use cookies.

Starting Session
-Start session: *session_start();*

-Setting variables: *$_SESSION['var_name'] = 'value';*
-Accessing variables: *$_SESSION['var_name'];*

Ending Session
-Syntax: *session_destroy( );*

Session ID
-get: *session_name()*;

-update: *session_name('new_name_str');*
  //if pass in str, then this is new name, and old name returned

Session Timeouts
-PHP runtime configuration holds details for various settings

-*session.gc_maxlifetime* is prop of this that holds value in seconds for how long data seen as garbage and cleaned up, including session data

-set runtime props via global function *ini_set('name', 'value')*
-For session timeout: *ini_set('session.gc_maxlifetime', 60 * 60 * 24)*

Session Security
-Though session data not passed from client to server, session ID is sent, thus exposing packet sniffing vulnerability

-As with sending passwords, only real way to provide security for this is to set page up to implement *SSL* for site

-To detect MITM, can save user ip in session variable via, then check & die, etc if ip changes:

```
$_SESSION['ip'] = $_SERVER['REMOTE_ADDR']
if( $_SESSION['ip'] != $_SERVER['REMOTE_ADDR')
   die();
```

-Can do similar check or combo check with *$_SERVER['HTTP_USER_AGENT']* which stores user browser info

Session Data Storage
-Session data stored in temp folder on webserver, set by PHP
-Can change where stored, by calling
  ini_set('session.save_path', '/home/user/mylocation')

-If running shared server, make sure to change default path
-Session data can also get large fast, so consider clearing folder now and then

JS & PHP Validations

Javascript Validation
-Better to do back-end, as front-end JS code can be edited as available to user
-Better thus to leave only basic validation to JS (*required* for form inputs, using proper types for <*input*> to check for num, phone, email, etc., etc.

-Use regex to check for required, not allowed, etc.

-Ex. form handling function calls validation function before submitting to back-end. If validation fails, shows error displaying reason for failure, else submits.

PHP Validation

-Basic validations: check for required *POST, GET,* etc. fields. Check for requirezd field format (ex. *something@place.com*). Check for proper length, range, type. Etc.

-After validating, should sterilize input before working with as seen in notes section ‘Preventing HTML Injection’ above

Composer

https://getcomposer.org/doc/

-Dependency mgmt tool for PHP that manages packages on a per-project basis, installing in a dir in project

-Inspired by NPM and Ruby’s bundler

Installation

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
```

```
php -r "if (hash_file('sha384', 'composer-setup.php') === 'c5b9b6d368201a9db6f74e2611495f369991b72d9c8cbd3ffbc63edff210eb73d46ffbfce88669ad33695ef77dc76976') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
```

```
sudo php composer-setup.php --install-dir=/bin --filename=composer
```

```
php -r "unlink('composer-setup.php');"
```

Basic Usage

-Dependencies for project stored in user created *composer.json* file, which describes dependencies and other metadata, stored in project directory

-Require via:

```
{
    “require”: {
        “package_name”: “version”
    }
}
```

-*package_name* is in form *“vendor_name/project_name”*, to allow multi projects w/ same name, but diff vendors

-ex. *igorw/json*   *saldaek/json*

-*"version"* can take in a *, allowing for flexible sub-version imports
  -ex. *"1.0.*"*   *//any version >= 1.0 && < 1.1*

Installing Dependencies
-Once dependencies defined in *composer.json,* install via *php composer.phar install* which installs to *vendor* directory in project dir

-On first install composer creates *composer.lock* file, which holds exact versions used by downloaded dependencies. Commit this.

-If *composer.lock* file is also present with *composer.json* during first install, same versions as in *lock* will be installed

-Update via *php composer.phar update* which also updates *composer.lock*

-Can also update specific dependencies only:
  php composer.phar update groupA/packF groupB/packY …

Packagist
-Default Composer package repo. Contains now just PHP packages, but JS, CSS, etc. also (DataTables, jQuery, etc.)

-Can search packages on site: https://packagist.org/

Autoloading
-Automatically loading PHP classes into a file without needing to explicitly stating *require, include*, etc. by defining function that *require* class then registers function as autoloader. Do a bunch of autoloading functions in one script, then just include that script to auto access all classes.

-Many Composer packages come with autoloading details. If so, Composer generates file *vendor/autoload.php.* As typically, simple *require* this file to have access to package classes
  require __DIR__ . '/vendor/autoload.php';

-Can also register own code to be autoloaded via *autoload* filed in *composer.json*

```
{
   "autoload": {
      "psr-4" : {"NamespaceX\\": "dirX/"}
   }
}
```

-If modify *composer.json* to include own autoloading, run *composer dump-autoload* to regenerate *autoload.php*

Libraries
-Any folder with a *composer.json* file is a package. To make the package a library, all it needs is a name

-Include name prop in *composer.json*: *“name”: “lain/my_package”*

-Give library a version with *composer.json* prop: *"version": "1.0.0"*

-Package can now be pushed to packagist, installed with composer, etc.

Advanced PHP

Namespaces

-Encapsulating items across projects with re-usable code to avoid collisions in classes, functions, constants, etc.

-In PHP, allow to group related classes, interfaces, functions, constants, etc.

-Declare with *namespace* keyword: *namespace myNamespace;* before definition of class/ script. All files in namespace will have this deceleration.

-Can also define sub-namespaces: *namespace myNamespace\sub\level;*

-After definition of namespace, that file exists within that namespace when included into a file. Can reference elements in multiple way:

-Call with full namespace via: *namespace\sub-namespace\my_variable*
ex. *$objDBController = new Database\DBController();*

-Can tell PHP to use a namespace for a class, etc. with
use path\to\file\some_functions;
use path\to\file\{someClass, anotherClass}’
-After, can just refer to class by *someClass,* etc.

-Can get current namespace with *namespace* keyword
$objDBController = new namespace\DBController();
namespace\print_database_name($objDBController);

-Can import class and call as alias with *as* keyword
require_once(‘dog.php’) //dog class
user \namespace\sub\file\dog as canine
$myDog = new canine();

-

Autoloading

-Automatically loading PHP classes into a file without needing to explicitly stating *require, include*, etc.$

-With autoloading, if try to create *new* class, by have not *included* class, autoloader will load class automatically

-Function works by first defining function inside script which takes in class and requires class inside function. Ex.

```
function loadView($class) {
    $path = $DOCUMENT_ROOT . '/views/';
    require_once $path . $class .'.php';}
  }
```

-Once have created autoload function, then register as autoloader by passing into global function *spl_autoload_register('function_name')*

-Then, repeat creation of autoloading functions and registrations with all desired classes to be included in that specific autloader, then *require_once my_autoload.php* in all file you want to have access to classes