PHP V

Package Managers

- The PHP community has largely relied on two package manager/ package repositories
  - PEAR
  - Composer/Packagist
- PEAR is the PHP Extension and Appplication repository
  - Was the system of choice for many years
  - Not commonly reccomended anymore

Composer

- Composer has become the PHP package manager of choice for most developers
- Is a PHP script itself, run from the command line
  - composer.phar is a PHP archive of executable code
- Configured using JSON file `composer.json`
- Dependencies installed using `php composer.phar install`

Composer.json

- The `composer.json` file is an object that uses many keys to specificy the configuration
- For pure package management, the most important key is `require`
- The value of require is an object whose keys are packages and whose values are package versions

```
{
    'require':{
          "ezyang/htmlpurifier" : ">0.0"
              }

}
```

Using Packages Installed with Composer

- Packages places all libraries installed inside the `vendor` directory in whatever directory it was run from
  - Each PHP project has its own `composer.json`
- When installing packages, composer updates its own autoloader script, so to include all libraries, just add

```
require_once __DIR__ . '/vendor/autoload.php';
```

Packagist

- The default repository used by Composer is Packagist
- Other repos can be specified by using the `repository` key in `composer.json`
- New packages are published by using `composer.json` with some additional keys
- A list of popular libraries can be found at https://phptrends.com/

In [ ]:

```
require_once __DIR__ . '/vendor/autoload.php';
```

In [ ]:

```
$faker = Faker\Factory::create();
echo $faker->name;
echo $faker->address;
echo $faker->text;
```

Frameworks

- Writing large webapps requires a lot of boilerplate code
    - Templating Engines
    - Making Database Requests
    - Organizing Codebases
    - Security
    - Processing Forms

Popular Frameworks

- As the community matures and best practices emerge, a few frameworks have bubbled to the top
  - CodeIgniter
  - CakePHP
  - Laravel

CodeIgniter Example

```
$this->load->database();
$query = $this->db->get('table_name');

foreach ($query->result() as $row)
{
        echo $row->title;
}
```

Debugging

- Logfiles
  - The first place you normally look for an error are the log files created by the PHP module of your webserver
  - There is one PHP module for everyone on GL, so for security reasons you can't access this
- Sending Errors to Client
  - Explicitly Printing
  - Telling PHP To report them

Var_Dump

- `var_dump` is a very similar command to `print_r`, but contains more information good for debugging
    - The object type is part of the output
    - The length of arrays are output

In [ ]:

```
$obj = json_decode('{"a":1,"b":2,"c":3,"d":4,"e":5}');
echo print_r($obj,true);
```

In [ ]:

```
var_dump($obj)
```

Printing Errors

- By default, any errors encountered during the execution of a PHP script are only written to the log, and not the screen
  - This is for security reasons
- To force error reporting use the function `ini_set`

  `

  ```
  ini_set("display_errors", 1);
  ```

- To control the level of errors that get reported, use the `error_reporting` function

  ```
  error_reporting(E_ALL);
  ```

Available to view at
https://www.csee.umbc.edu/~bwilk1/433/php_examples/errors_on.php

```
<?php

    error_reporting(E_ALL);
    ini_set("display_errors", 1);

    $aliens_title_by_release_year[1979] = "Alien";                # good
    $aliens_title_by_release_year[1986] = "Aliens";               # good
    $aliens_title_by_release_year[1992] = "Alien 3";              # eh...
    $aliens_title_by_release_year[1997] = "Alien: Resurrection";  # avoid
?>
    <h1>Warnings & Errors On</h1>
    <ul>
<?
    foreach(array_keys($aliens_title_by_release_year) as $key) {
        print "<li>$key: $aliens_title_by_relaese_year[$key]</li>";
    }
?>
    </ul>
```

Security

- PHP is one of, if not the most popular point to start a cyberattack
- PHP provides an interface between strangers and your server
- It is imperative you consider the security implications of your PHP script, not just for you, but for other users of your page

Password Hashing

- PHP is commonly used to build log in systems
- Storing users passwords is a serious responsibility
- Using frameworks is one way to prevent mistakes
- The other is to use the built in functions `password_hash` and `password_verify`
  - Requires PHP 5.5+ aka not GL

In [ ]:

```
$my_password = 'password1234';
echo password_hash($my_password,PASSWORD_DEFAULT);
echo password_hash($my_password,PASSWORD_DEFAULT);
```

In [ ]:

```
$hash1 = password_hash($my_password,PASSWORD_DEFAULT);
echo password_verify($my_password,$hash1);
echo password_verify('password',$hash1);
```

Input Validation

- A major vulerability is also one of PHP's greatest strengths
  - Accepting user input
- We will talk about a few very specific issues in a bit, but in general you should always verify your input to be what you expect it to be
- PHP has two methods to help with this
  - filter_var
  - filter_input

Filtering

- Input filtering is checking if the input given matches the expected format
- The two methods mentioned before use constant to check the input
    - This is only a first pass, more specific filtering should be done based on your app needs
- Common filters
    - FILTER_VALIDATE_EMAIL
    - FILTER_VALIDATE_URL
    - FILTER_VALIDATE_IP

In [ ]:

```
$my_email = "bryan.wilkinson@umbc.edu";
filter_var($my_email,FILTER_VALIDATE_EMAIL);
```

In [ ]:

```
$my_bad_email = "<script>doEvil();</script>@umbc.edu";
filter_var($my_bad_email,FILTER_VALIDATE_EMAIL);
```

Sanitizing

- An alternative to filtering is to sanatize your input
- This removes harmful characters from the input
    - It is considered less secure, because if an attacker knows this is happening, they can try an be clever to get around it
- Better to reject if you can

Sanitizing

- Santitizing uses the same functions, but with different constants
- Common constants for sanitization are
    - FILTER_SANITIZE_EMAIL
    - FILTER_SANITIZE_URL
    - FILTER_SANITIZE_STRING

In [ ]:

```
$my_email = "bryan.wilkinson@umbc.edu";
$my_bad_email = "&34;script>doEvil();</script>@umbc.edu";
echo filter_var($my_email,FILTER_SANITIZE_EMAIL);
echo filter_var($my_bad_email,FILTER_SANITIZE_EMAIL);
```

Network Request Security

- Another common vulerability is to trust encrpyted data, which is still vulernable to man-in-the-middle attacks
- To prevent against this, you should explicity tell PHP to ensure that the response of the network request is from the server requested

```
$context = stream_context_create(array('ssl' => array('verify_peer' => T
RUE)));
$body = file_get_contents('https://api.example.com/search?q=sphinx', fal
se, $context);
```

In [ ]:

```
$context = stream_context_create(
array('ssl' => array('verify_peer' => TRUE)));
$body = file_get_contents('https://www.umbc.edu', false, $context);
```

SQL Injections

- We didn't cover using PHP with databases in this course, but it is a very common use of them
- Using user input directly in an SQL query is a very bad idea
  - Can leak data
  - Can leak information about your database set up
- Steps to reduce
  - Valdiate data first
  - Escape input or use prepared statements

Example from https://php.earth/docs/security/sql-injection

```
$query = "SELECT username, email FROM users WHERE id = ?";

$stmt = $mysqli->stmt_init();

if ($stmt->prepare($query)) {
    $stmt->bind_param("i", $id);
    $stmt->execute();
    $result = $stmt->get_result();
    while ($row = $result->fetch_array(MYSQLI_NUM)) {
        printf ("%s (%s)\n", $row[0], $row[1]);
    }
}
```

Code Injection

- Another danger is someone including their own PHP into your code
- This can happen when:
  - `eval` is used
  - A user input is passed to `include` or `require`
  - A file name is passed to open

Preventing Code Injection

- Don't use eval
- If you are doing dynamic includes, use a switch statement or something, and don't directly just use the variable
- Validate your filenames, and don't use a filename directly

Directory Traversal

- Another danger about files and file names is directory traversal
- A malicous user could send the file name `/` or `../` and get somewehre they shouldn't
  - Explicity check for filenames start with this
  - Run standard filtering and sanitation on file names
  - Don't use the user supplied file names!

Cross Site Scripting

- A major danger to users of your site is cross-site scripting
  - If your database is comporimised, it could be placed there
  - It could be done through public content through a form like comments
- Escape escape escape
  - If you are sending user generated content back to the client, use HTMLPurify library

In [ ]:

```
$untrustedHtml = "<script><iframe src=''></script><b>Hello</b>";
$config = HTMLPurifier_Config::createDefault();
$config->set('HTML.Allowed', 'p,b,a[href],i'); // basic formatting and links
$sanitiser = new HTMLPurifier($config);
$output = $sanitiser->purify($untrustedHtml);
```