PHP + MySQL

- MySQL on the command line is great and all... well not its not really that great
- Using MySQL in PHP is somewhat similar to the command line:
  - Set up a connection to a MySQL database
  - Issue a bunch of commands to the database

PDO

- PHP Data Objects
- The modern way to access databases from within PHP
  - No more `mysql_connect`, `mysql_query`, etc.
  - No, the `mysqli` commands aren’t really any better.

PDO Connection

- Still need the same pieces of data:
  - Database host
  - Username
  - Password
PDO Connection

- We make a new PDO object based off the data source properties
- Can make PDO objects for a wide variety of databases, not just MySQL

```php
$dsn = 'mysql:dbname=cs337;host=localhost';
$user = 'root';
$password = 'somepassword';
$db = new PDO($dsn, $user, $password);
```

PDO Connection

- For our AWS Servers, access is only available from localhost, and no user/password is required

```php
$dsn = 'mysql:dbname=cs337;host=localhost';
$db = new PDO($dsn);
```

- Once we have a connection set up, we can start talking to our database using our newly created object

```php
<?php
$dsn = 'mysql:dbname=cs337;host=localhost';
$user = 'root';
$password = 'somepassword';
$db = new PDO($dsn, $user, $password);

// Get the submitted form data
$name = $_REQUEST['name'];
$phone = $_REQUEST['phone'];
$email = $_REQUEST['email'];

// Create our insert query
$sql = "INSERT INTO staff (name, phone, email) VALUES ("' . $name . '","' . $phone . '","' . $email . '");
$db->query($sql);
```
Aside: PHP Strings & Variable Expansion

```php
// Create our insert query
$sql = 'INSERT INTO staff (name, phone, email) VALUES (' . $name . ', ' . $phone . ', ' . $email . ');';
```

- Here we have a PHP string surrounded by double quotes.
- Inside, we have variables $name, $phone, $email
- These will be replaced with their actual string contents.
- The curly braces {} help PHP limit variable name searching.

Aside: PHP Strings & Variable Expansion

- Variable expansion only happens inside double quoted strings
- Single quoted strings are evaluated as literals

```
<?php
ini_set('display_errors', 'on');
error_reporting(E_ERROR | E_WARNING | E_NOTICE | E_PARSE);

$height = 100;
echo "$heightpx";
echo "$in";
echo "$heightpx";
echo "$in";
echo "$heightpx";
echo "$in";
echo "$heightpx";
echo "$in"
```

Congratulations!

You now know just enough to be very dangerous…
Security Concerns

- Trusting user input is very dangerous
- SQL Injection and Code Injection
- Cross Site Scripting attacks
- Examples

Prepared Statements

- Allows us to make sure that nothing can ‘break out’ of the SQL statement.
- Much more secure than trying to build SQL statements through string concatenation.
- If you encounter `mysql_query` or `mysqli_query`, you should really consider refactoring to use PDO.
```php
ini_set('display_errors', 'on');

$dsn = 'mysql:dbname=cs337;host=localhost';
$user = 'root';
$password = 'password';
$db = new PDO($dsn, $user, $password);

$sql = "SELECT * FROM staff
WHERE phone=? AND name=?";

$stmt = $db->prepare($sql);
$stmt->execute(array("626-1541", "Jan"));
$results = $stmt->fetchAll(PDO::FETCH_CLASS);
print_r($results);
```

Prepared Statements

- We call the PDO::prepare() method first
- This returns a new PDOStatement object
- We then call the execute() method on the newly created PDOStatement, not on the PDO object


```php
$stmt = $db->prepare($sql);
$stmt->execute(array("626-1541", "Jan"));
```

- We then call the execute() method on the newly created PDOStatement, not on the PDO object
- We pass along an array of replacement values in an array to the execute method
- The order of the array values must match the SQL

Prepared Statements

- Note that you do not enclose the ? placeholders in single quotes
- The PDO layer and database takes care of quoting strings for us

```php
$sql = "SELECT * FROM staff
      WHERE phone=? AND name=?";
```

```php
$sql = "INSERT INTO staff (name, phone, email) VALUES ('{$name}','{$phone}','{$email}')";
```

---

PHP Objects

Round Two

---

More Object-y Things

- OOP - Object Oriented Programming
- PHP supports just about all OOP patterns
- Static Object calls vs Instantiated
Inheritance

• Basically, Class A can inherit from Class B
• Define properties and behavior on a “Parent” class which can be inherited by “Child” classes.
• Example

```php
<?php

class droid
{
    private $name = "";
    public function __construct($setName)
    {
        $this->name = $setName;
    }
    public function status()
    {
        echo "I'm \$this->name the ", get_class($this) . ":\n";
    }
}

class protocolDroid extends droid
{
    public function translate()
    {
        return "Beep boop";
    }
}

class astromechDroid extends droid
{
    public function pilot()
    {
        return "Zzzooooooom!";
    }
}
$c3po = new protocolDroid("C3PO");
c3po->status();
$r2 = new astromechDroid("R2D2");
r2->status();

• The droid class defines a status() method.
```
Inheritance

- Inheritance is the big idea.
- PHP implements this via the `extends` keyword.
- Here the `protocolDroid` class `extends` the `droid` class.

```php
<?php

class droid {
    private $name = "";
    public function __construct($setName) {
        $this->name = $setName;
    }
    public function status() {
        echo "I'm " . $this->name . " the " . get_class($this) . ".\n";
    }
}

class protocolDroid extends droid {
    public function translate() {
        return "Beep boop";
    }
}

• When one class `extends` another, it is inheriting the properties and methods of the parent class.

• When a Child class `extends` a Parent class, the Child class `inherits` the methods and properties of the Parent.

• (that sounds suspiciously like something that may turn up on a final)

• Here the `protocolDroid` class will have a `status()` method, even though it doesn't define it itself.
Inheritance

- The `get_class()` PHP function returns a string containing the name of the class.
- The Child classes do not implement their own constructor, so the Parent’s is used.

Inheritance Demo

```
php/inheritance.php
```

Encapsulation

- Fancy way of saying “hiding things from people”
- Allows the developer of a Class a way to keep the implementation details of the Class hidden from the outside of that Class.
- Allows for selective inheritance.
Encapsulation Case Study

- Suppose we have a Class describing a Ticketing service.
- Our Ticketing service can create a support ticket, update a ticket, retrieve a ticket, etc.

Ticket Example

```php
class ticketer {
    // Property to hold our database connection
    public $db;

    public function __construct() {
        // Connect to our database
        $this->db = new PDO($dsn, $user, $pass);
    }

    public function newTicket() {
        $sql = "INSERT INTO tickets ....";
        $stmt = $this->db->prepare($sql);
        $stmt->execute();
        $newTicketID = $this->db->lastInsertID();
        return $this->getTicket($newTicketID);
    }

    public function getTicket($ticketID) {
        // ...
    }
}
```

Ticket Example

```php
require "ticket_class.php";
$tickets = new ticketer();
$newTicket = $tickets->newTicket();
```

- A sample bit of code that uses our ticketer class
- Creates a new instance of our ticketed class.
- Creates a new ticket.
Ticket Example

- We want to do some additional querying that’s not built into the ticketer class
- Grab the ticketer::$db property from our object.
- Execute our own local SQL queries.

```php
<?php
require "ticket_class.php";
$tickets = new ticketer();
$newTicket = $tickets->newTicket();
$ticketDB = $tickets->db;
$sql = "SELECT * FROM tickets WHERE ...";
$stmt = $ticketDB->prepare($sql);
$stmt->execute();
$results = $stmt->fetchAll();

(...)
```

Ticket Example

- Alice decides MySQL was too slow
- Switched to Redis for our data store backend.

```php
class ticketer {
    // Property to hold our database connection
    public $db;
    ...
}
```

http://redis.io

Ticket Example

- What happens to our code that depended on getting a reference to the database connection?

```php
<?php
require "ticket_class.php";
$tickets = new ticketer();
$newTicket = $tickets->newTicket();
$ticketDB = $tickets->db;
$sql = "SELECT * FROM tickets WHERE ...";
$stmt = $ticketDB->prepare($sql);
$stmt->execute();
$results = $stmt->fetchAll();

(...)
```
visibility

- PHP gives us tools to prevent access to properties and methods from outside of the object itself.
- This is known as visibility
  - public
  - private
  - protected


public

- Public properties and methods are available to any code that references the class or instantiated objects.
- This is why we were able to get a reference to the ticketer database property.

```php
class ticketer {
    // Property to hold our database connection
    public $db;
    ...
}
```

```php
require "ticket_class.php";
$tickets = new ticketer();
$newTicket = $tickets->newTicket();
$ticketDB = $tickets->db;
$sql = "SELECT * FROM tickets WHERE ..."
```

private

- I lied a little bit back there when we talked about inheritance
- Private properties and methods are only available within the object instances itself.
- This would prevent anyone from getting a reference to the ticketer database property.

```php
class ticketer {
    // Property to hold our database connection
    private $db;
    ...
}
```

This would cause a fatal error now
• Protected properties and methods are available only within the object instances itself and any subclasses.

```php
<?php
    class droid {
        protected $name = ""
        public function __construct($setName) {
            $this->name = $setName;
        }
        public function status() {
            echo "I'm {$this->name} the ".
            get_class($this) . ".\n";
        }
    }
    class astromechDroid extends droid {
        public function pilot() {
            return "Zzzooooooom!"
        }
    }
    $r2 = new astromechDroid("R2D2");
    echo $r2->description()
    . "\n";
    echo $r2->name . "\n";
    ```

Static Access

• Up to now we’ve mostly been instantiating our classes as objects

• But we don’t have to!

• Maybe you don’t want a whole bunch of distinct objects, maybe you want a utility class?
Static Access

- Using the static keyword

```php
ini_set('display_errors', 'on');

class util {
    public static function pow($base, $power) {
        $product = 1;
        for ($i = 0; $i < $power; $i++) {
            $product = $product * $base;
        }
        return $product;
    }
}

echo util::pow(2, 8) . "\n";
```

- Using the `className::method()` syntax we can call a static method directly from the Class definition without having to create an instance of that Class.

- Can also access static properties in a similar way.

- Also used to reference constants on Classes.

Constants

```php
class util {
    const HOSTNAME = 'localhost';
    const CURRENT_VERSION = '1.7.10';
}

echo util::CURRENT_VERSION . "\n";
```

- Classes can define constants

- Constants cannot be modified at runtime

- Good for things you know won't change, like a version number or other setting.
Working with JSON

- PHP has built in support for dealing with JSON encoded data
- Convert JSON text to PHP data structures:
  - $var1 = json_decode( string );
- Convert PHP data structures to JSON
  - $json = json_encode( $var1 );
- Examples