PROJECT #1: “Auction Site” v1.0

PART 4: “Adding the Monolog Package”

Objectives:

- Install and configure the Monolog package
- Add code to log failures in the Database class

Instructions:

It’s time to add the logging feature to your Auction application.

You could spend countless hours trying to create your own logger or simply use a popular third party library called Monolog.

Monolog is a full featured logger for PHP which boasts the ability to send log files to files, sockets, inboxes, databases and various other web services.

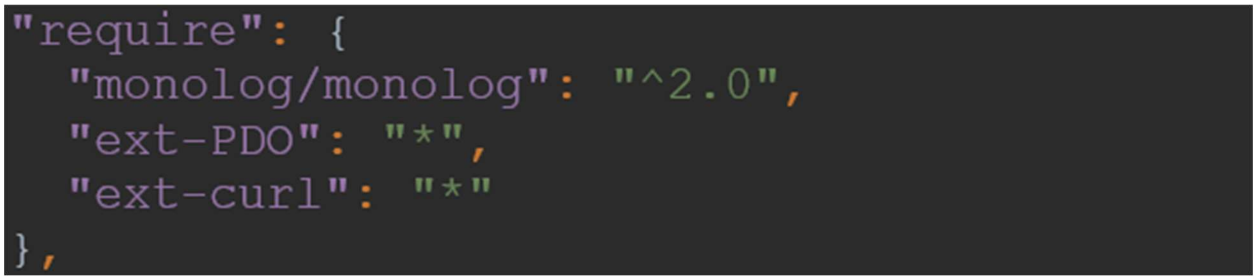
It has been downloaded over 110 million times and is currently the most popular third party logging utility available in PHP. To read more about its implementation and features, visit the official website at https://github.com/Seldaek/monolog or on Packagist at https://packagist.org/packages/monolog/monolog

Monolog works by defining “channels” and log levels. Channels define how you want your messages logged (email, file, database, etc.). Monolog defines 8 different log levels: DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL, ALERT & EMERGENCY all based upon the RFC 5424.

More information including information about the difference between each log level can be found here: https://github.com/Seldaek/monolog/blob/master/doc/01-usage.md#log-levels

1. To add the Monolog package to your existing project, you simply need to add it to the existing **composer.json** file and update the dependency list.

2. Open the **composer.json** file in the root of the application and add the following line just below the **"require"** section:



```
"require": {
  "monolog/monolog": "^2.0",
  "ext-PDO": "*",
  "ext-curl": "*"
},
```

3. Next, you need to tell Composer to update the package list. To do this, drop to the command prompt (Start a new SSH session) and change into the project directory.

4. Issue the following command to update the list of required packages:

`composer update`

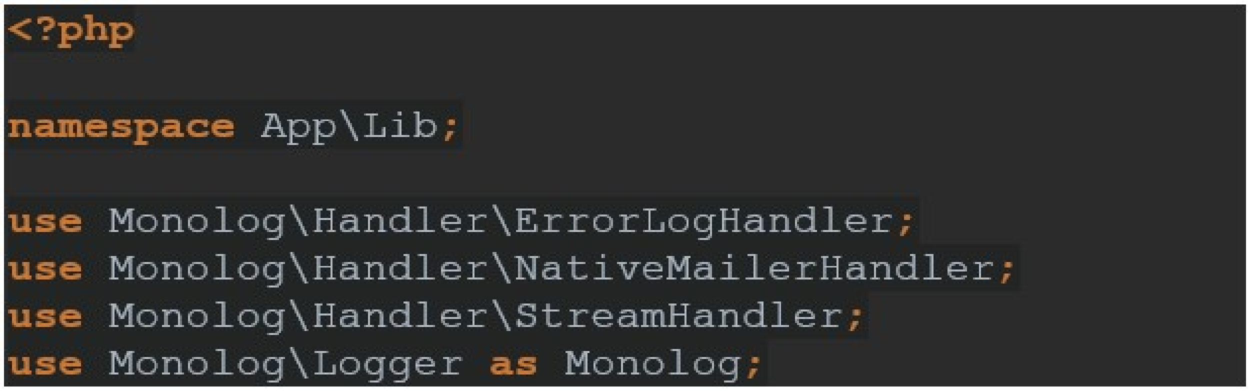
```
[auction]$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing psr/log (1.0.2): Loading from cache
  - Installing monolog/monolog (2.3.0): Loading from cache
...
...
Writing lock file
Generating autoload files
[auction]$
```

Composer will automatically regenerate the autload file to include the files required for the Monolog package.

5. Before we can use the Monolog package, we first need to define a **Logger** class with some helper methods.

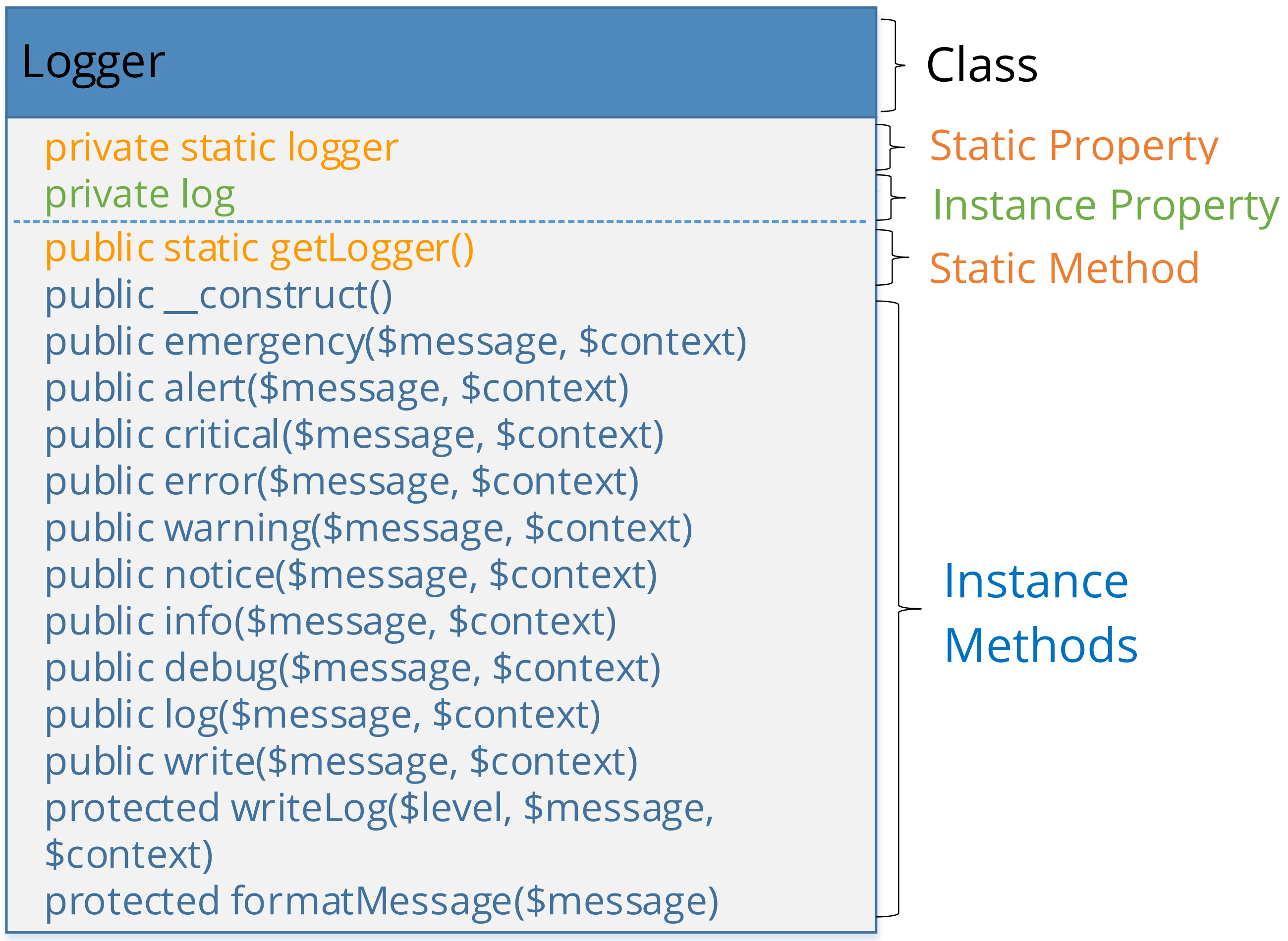
   Create a new php class file in the **app/Lib/** directory called **Logger.php**

6. The **Logger** class will reside in the **App\Lib** namespace and use the following list of classes:



```php
<?php

namespace App\Lib;

use Monolog\Handler\ErrorLogHandler;
use Monolog\Handler\NativeMailerHandler;
use Monolog\Handler\StreamHandler;
use Monolog\Logger as Monolog;
```

In the **Logger** class definition, you will create the following properties and instance methods based on the UML diagram below:


Logger
Class
private static logger
Static Property
private log
Instance Property
public static getLogger()
Static Method
public __construct()
public emergency($message, $context)
public alert($message, $context)
public critical($message, $context)
public error($message, $context)
public warning($message, $context)
public notice($message, $context)
public info($message, $context)
public debug($message, $context)
public log($message, $context)
public write($message, $context)
protected writeLog($level, $message, $context)
protected formatMessage($message)
Instance Methods

The **Logger** class contains only one static property; a logger object. The Logger class also contains one instance property; the log context. Notice that the visibility is set to **private** for both properties.

The logger class contains the following methods:

- A static method called **getLogger**. This method is used to retrieve a new/existing instance of the Logger class.

- A constructor method.

- The following instance methods: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, **debug**, **log** & **write**. These methods all have similar behaviour: they all forward a message to the **writeLog** method with varying severity.

- A method called **writeLog**. This is a helper method which formats incoming messages before passing it onto the Monolog logger to be recorded into the log channels.

- A method called **formatMessage**. This is a helper method used to format the incoming message so it can be neatly written to the logger.

Defining Logger Class Properties

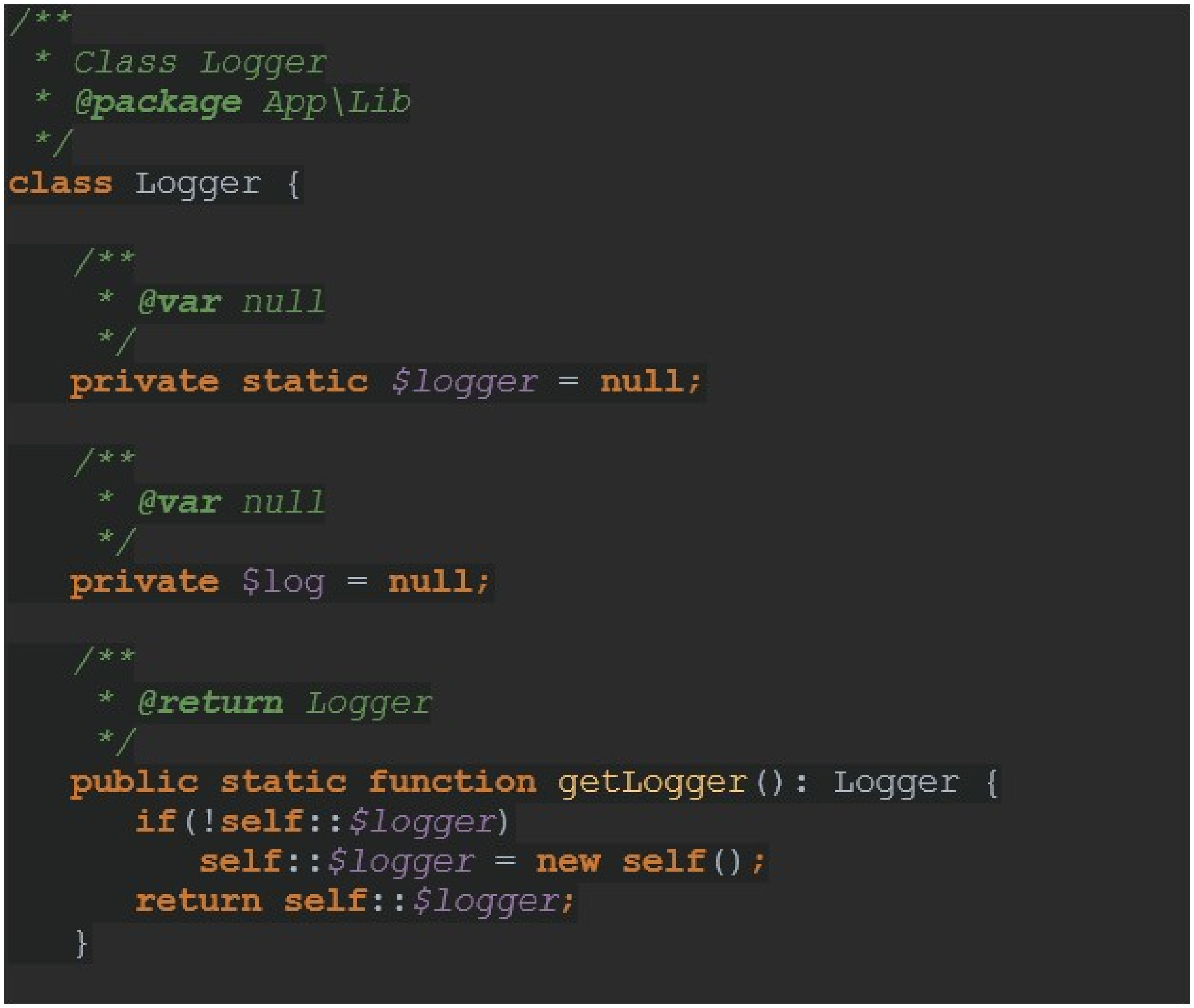
7. Within the **Logger** class definition, create the private static variable as illustrated in the UML diagram.

8. Create a private instance property as illustrated in the UML diagram. This will contain the Monolog instance.

Defining Logger Class Methods

Method: getLogger() : Logger

9. Within the **Logger** class definition, create the public static method named **getLogger**. This function uses a common PHP design pattern called the singleton pattern. It's purpose is to retrieve the active Monolog instance. If no instance exists, it will make a new one.



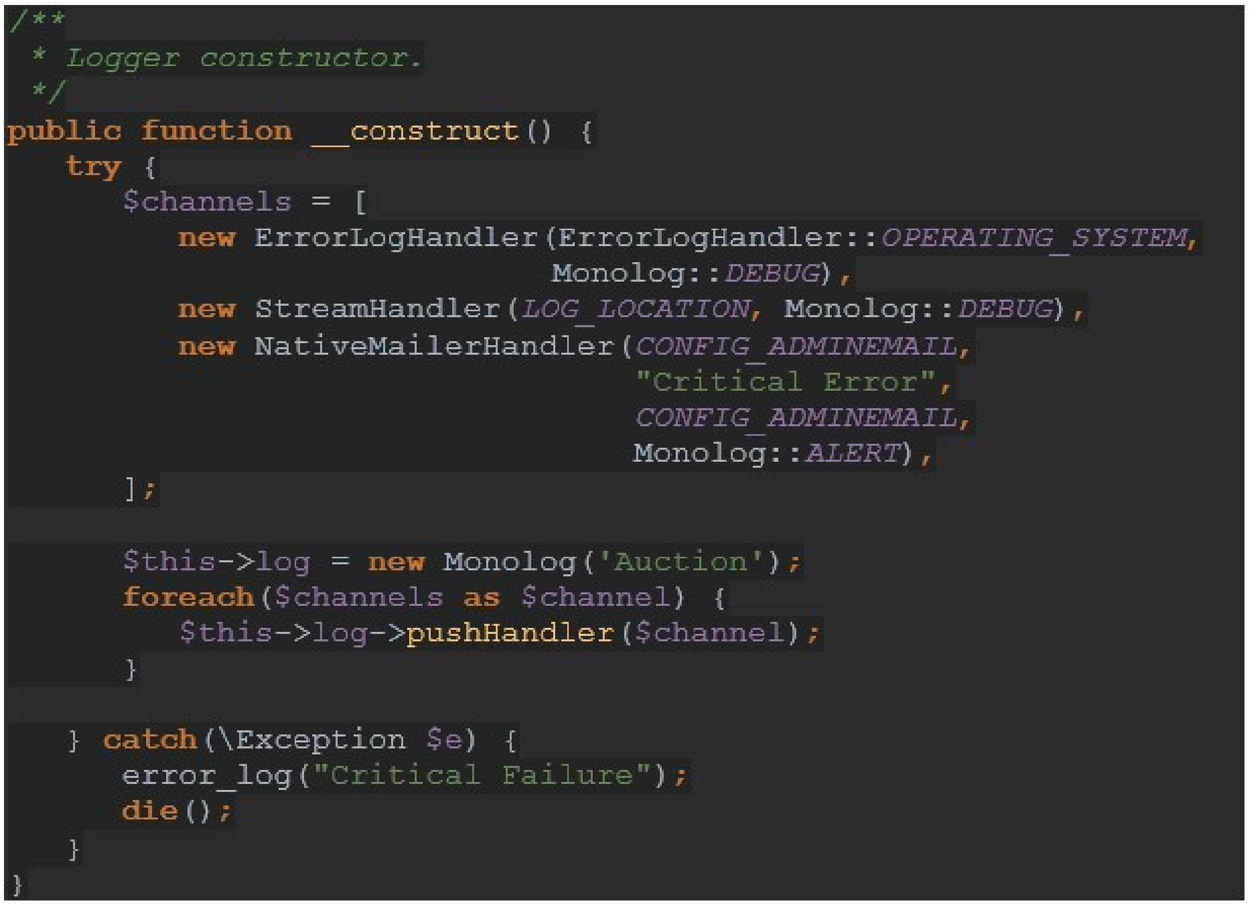
```php
/**
 * Class Logger
 * @package App\Lib
 */
class Logger {

   /**
     * @var null
     */
   private static $logger = null;

   /**
     * @var null
     */
   private $log = null;

   /**
     * @return Logger
     */
   public static function getLogger(): Logger {
      if(!self::$logger)
         self::$logger = new self();
      return self::$logger;
   }
```

Method: __construct()

10. Within the **Logger** class definition, create a public **constructor** method. The purpose of this method is to create the Monolog instance with the appropriate channels and store it in the **$log** instance variable.

    To access the object, you will use the **getLogger** method you created earlier.

11. Use the following code to define 3 channels you will log to: the **ErrorLogHandler** which logs to the **error_log** file located in your user directory's log directory. The **StreamHandler**, used to log to the application's log directory. The **NativeMailerHandler** used to log via email address.

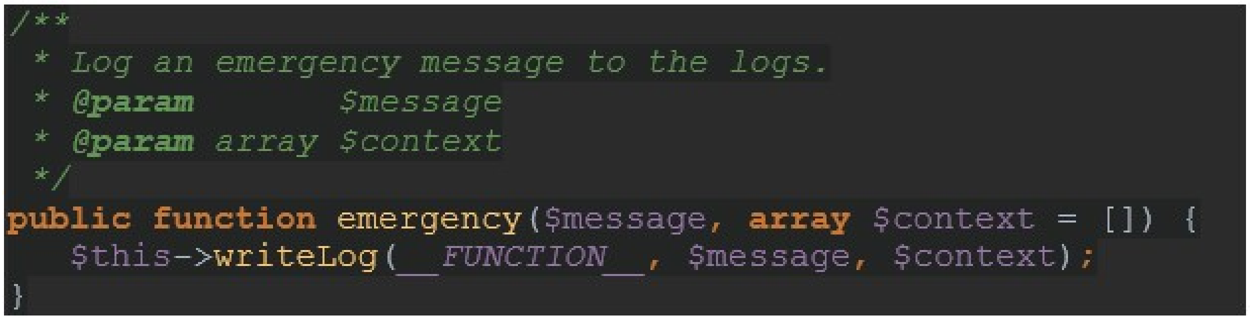


```
/**
 * Logger constructor.
 */
public function __construct() {
    try {
        $channels = [
            new ErrorLogHandler(ErrorLogHandler::OPERATING_SYSTEM,
                                Monolog::DEBUG),
            new StreamHandler(LOG_LOCATION, Monolog::DEBUG),
            new NativeMailerHandler(CONFIG_ADMINEMAIL,
                                    "Critical Error",
                                    CONFIG_ADMINEMAIL,
                                    Monolog::ALERT),
        ];

        $this->log = new Monolog('Auction');
        foreach($channels as $channel) {
            $this->log->pushHandler($channel);
        }

    } catch(\Exception $e) {
        error_log("Critical Failure");
        die();
    }
}
```

Methods: emergency, alert, critical, error, warning, notice, info, debug

12. Create a new method for each of the **8** possible log levels. All methods have the same behaviour but vary on level of severity. For each of the following methods: **emergency**, **alert**,

critical, **error**, **warning**, **notice**, **info** and **debug**, add the following code. See example below:

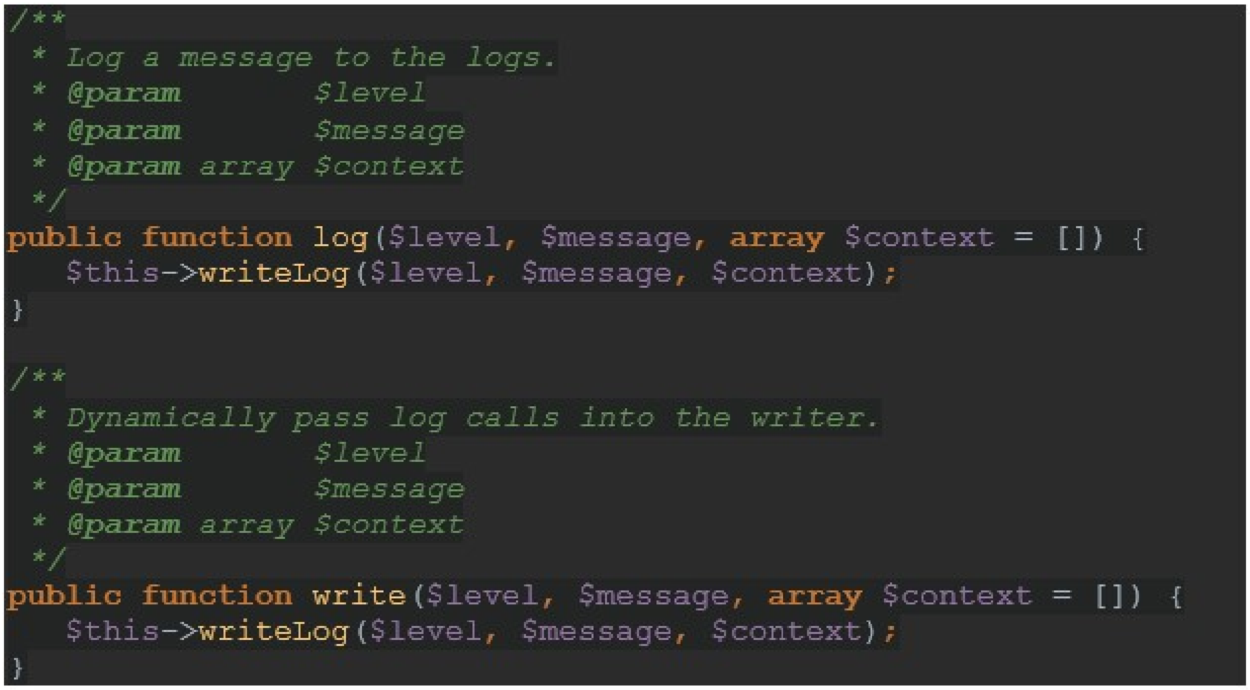


```
/**
 * Log an emergency message to the logs.
 * @param       $message
 * @param array $context
 */
public function emergency($message, array $context = []) {
   $this->writeLog(__FUNCTION__, $message, $context);
}
```

Copy this code and change the method name and phpdoc block for each of the 8 log levels.

Methods: log, write

13. Create two new methods, **log** and **write**. Both of these methods have the same behaviour and are helper methods used to forward a message through to the Monolog instance.



```php
/**
 * Log a message to the logs.
 * @param       $level
 * @param       $message
 * @param array $context
 */
public function log($level, $message, array $context = []) {
    $this->writeLog($level, $message, $context);
}

/**
 * Dynamically pass log calls into the writer.
 * @param       $level
 * @param       $message
 * @param array $context
 */
public function write($level, $message, array $context = []) {
    $this->writeLog($level, $message, $context);
}
```

Method: writeLog

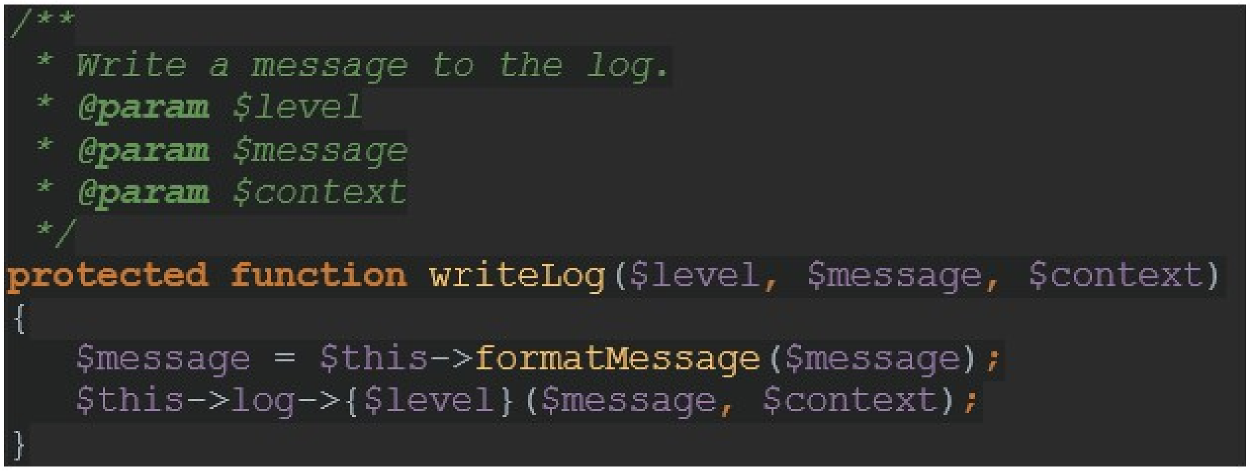
Create a public method called **writeLog**. This method's purpose will be to call the formatMessage helper method before forwarding it on to the Monolog channels defined in the constructor.

The method has three parameters:

- A string parameter named **$level**. This corresponds to one of the eight log levels defined earlier.
- A string parameter name **$message**. This is the message to be logged to the Monolog channel(s).
- An array paramter named **$context**. This allows you to provide additional information regarding the context of the message.

14. In the method, call the helper method **formatMessage**, passing in the message from the parameter. Save the return string into a new variable called **$message**.

15. Next, call the property **$log**'s **$level** method passing in the formatted message and the context.



```php
/**
 * Write a message to the log.
 * @param $level
 * @param $message
 * @param $context
 */
protected function writeLog($level, $message, $context)
{
   $message = $this->formatMessage($message);
   $this->log->{$level}($message, $context);
}
```

Method: formatMessage($message)

Create a public method called **formatMessage**. This method's purpose will be format incoming messages so they may be output to the channels properly.

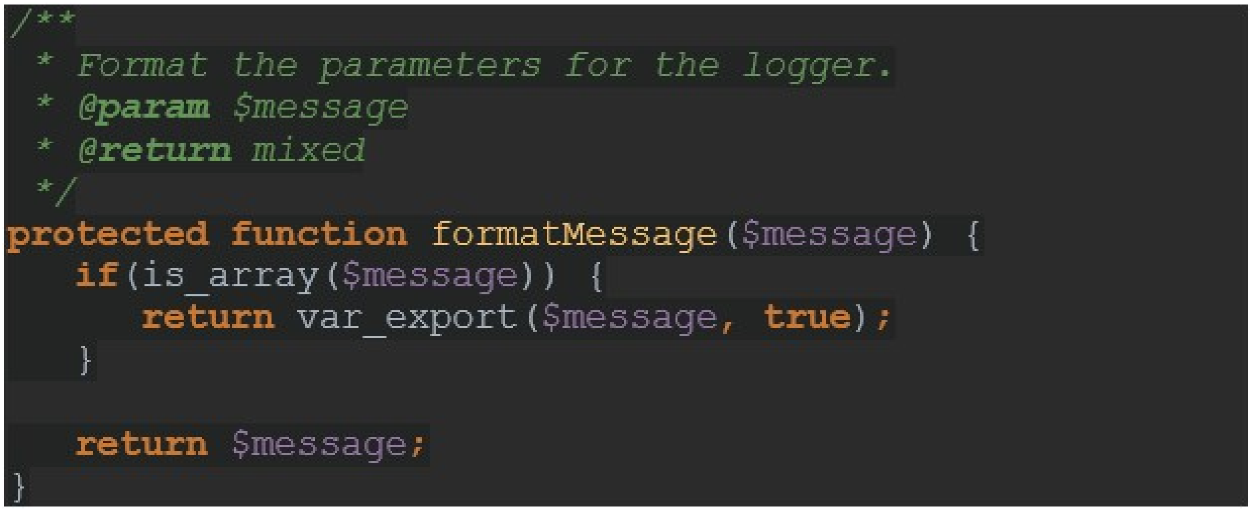
The method has one parameter:

- A string parameter name **$message**. This is the message to be logged to the Monolog channel(s).

16. In the method, create a conditional statement (**if**) , which checks if the **$message** parameter is an array.

    a. If the conditional statement is a boolean ***true***, call the **var_export** function and pass it two arguments. First, the **$message** and second, a boolean ***true***. Return the results of the **var_export** function call.

    b. If the conditional statement is a boolean ***false***, simply return the **$message** parameter.



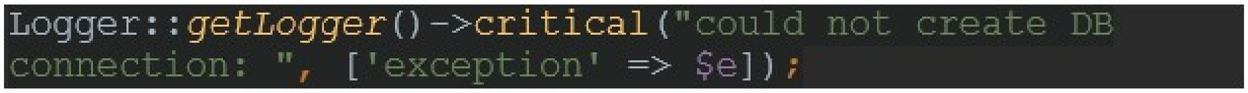
```
/**
 * Format the parameters for the logger.
 * @param $message
 * @return mixed
 */
protected function formatMessage($message) {
   if(is_array($message)) {
      return var_export($message, true);
   }

   return $message;
}
```

The **Logger** class is now complete. The next step is to test it. To do this, you will modify the **Database** class to log messages if there is an exception thrown.

This will make it extremely easy to trackdown any bugs or issues while developing the rest of the project.

17. Re-open the **Database** class file and locate the **constructor** method.

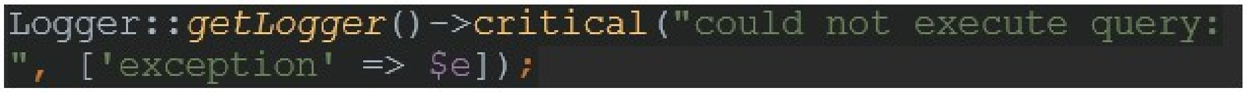
18. Within the **catch()** portion of the method body, add the following code just before the **die()** method:



```
Logger::getLogger()->critical("could not create DB
connection: ", ['exception' => $e]);
```

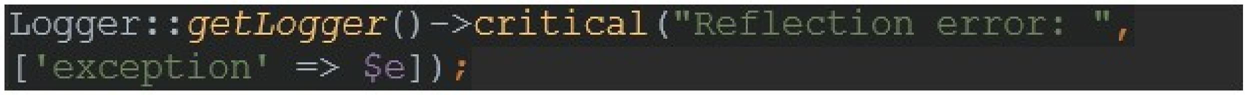
This force the application to log a new critical message to the Monolog channels when there is a failure to connect to the database.

19. Within the **sqlQuery** method, add the following code to the **catch** section directly above the **die()** method call:



```
Logger::getLogger()->critical("could not execute query:
", ['exception' => $e]);
```

20. Within the **fetch** method, add the following code to the **catch** section directly above the **die()** method call:



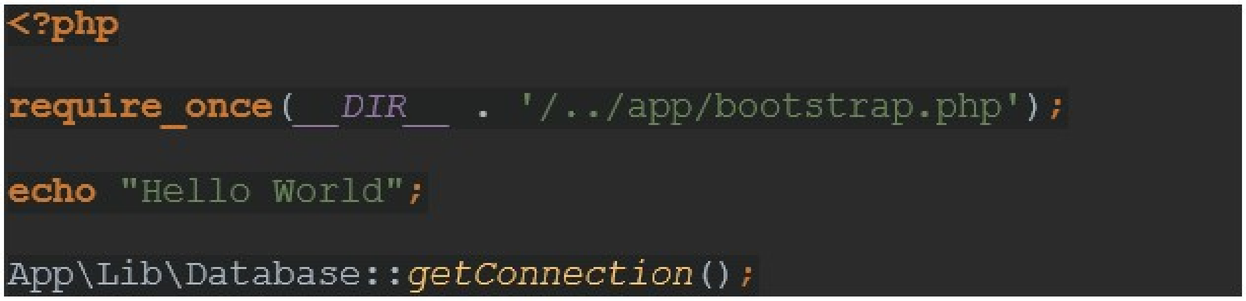
```
Logger::getLogger()->critical("Reflection error: ",
['exception' => $e]);
```

21. To test that the Logger is working correctly, open your **config.php** file in the **app\Config** directory.

22. Temporarily change your database password to something incorrect such as '**asdf**' to force the application to throw an

exception and therefore log the error message. Save and upload the file.

23. Open the **index.php** file you created earlier and add the following code to call the **Database** class:

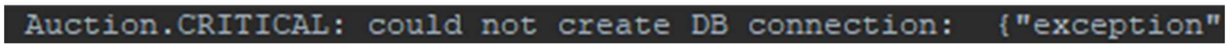


```
<?php
require_once(__DIR__ . '/../app/bootstrap.php');
echo "Hello World";
App\Lib\Database::getConnection();
```

24. Save and upload the **index.php** file and open it up in your browser. You should not see any error output to the screen.

25. Back in PHPStorm, open the **app\logs\app.log** file on the **Remote Host**. (If you do not see the file, click on the refresh button at the top of the window.)

    You should see the error which was thrown by the Logger:



```
Auction.CRITICAL: could not create DB connection:  {"exception"
```

26. Remember to change your password back in the config.php file

You're now ready to move on to the next section.