COSC260 Assignment 3: Server-Side Quiz Application

Your task is to develop a basic server-side user registration service in PHP. This service will operate in a very similar manner to the one your client-side application interacted with in Assignment 2.

Upon receiving a HTTP POST request, with the fields `name`, `age`, `email` and (optionally) `phone`, your service should return a user ID (randomly generated) to the client.
All requests must be checked for invalid input, and if found, an error message and appropriate HTTP status code must be returned. All successful requests should be written to a file, in JSON format.

There are 5 components to the assignment:

A) **Error response function**
B) **Request validation**
C) **Data validation**
D) **User ID**
E) **Database**

Starter Code

You are provided with starter code, both for your server-side application, and for a client-side testing application. Please use the PHP starter code provided, as you will be required to use some of the pre-existing variables present. The client-side application is used to test your server-side code, as it allows you to make a POST request using a form, with output displayed. **Your submission for Assignment 3 will contain server-side code only**.

Deploying Your Code

This is also covered in Tutorial 7 (Week 10)
In order to test your PHP code, you can either run your own web server, or use Turing which has everything you need installed. To use Turing for web development, simply make a folder (if it doesn't already exist) in your Turing home directory named `public_html`
Any files placed in this folder will be accessible via the URL:

https://turing.une.edu.au/~username/file.php (change 'username' and 'file' to suit)

The UNIX permissions on your PHP files **MUST** allow for **other** users to **read**, since the web server process is running on Turing as a different user. You can enable an autoindex file listing by giving **other** users **execute** permissions on the **folder**. (+x on a directory allows for listing of folder contents)

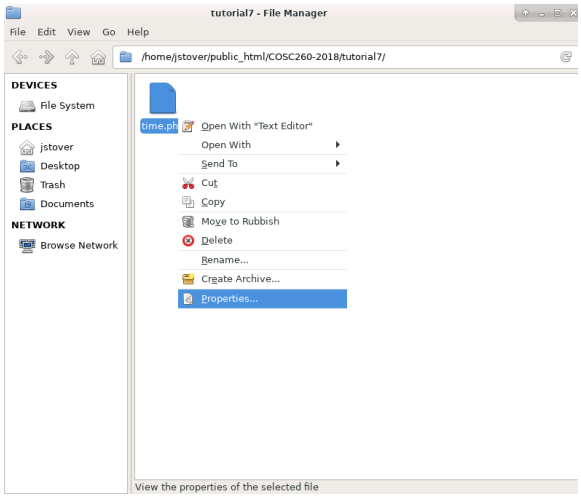
You can do this via the command line:

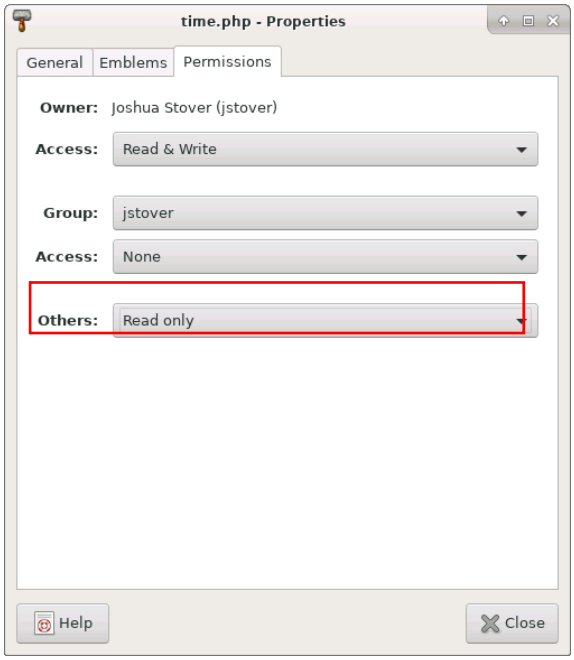
```
chmod a+r ~/public_html/myfile.php
```
*Allow **all** to **read** the file **myfile.php***

```
chmod a+x ~/public_html
```
*Allow **all** to **execute** the folder **~/public_html***

Or by using the turing desktop GUI (process is the same for **execute** permissions):

- Right click on the file you want to modify -> **Properties**
- Go to **Permissions** tab and change the **Others** setting to **Read only**





Testing Your Service

You are provided with a test client (HTML, CSS, and JS with jQuery) which allows you to submit data to your web server, using a web form: you can use this to test your server-side application. The client will display responses from the server both on the page and in the JS console. To use this testing client, you will need to change the `url` variable at the start of the `submit.js` file to the address of your web server.
If you wish to use this client for testing, please familiarize yourself with the code, and feel free to edit it if needed.

Make sure to also perform your own testing and not rely solely on the provided client. You should also use the 'network' and 'console' tabs of your browser developer tools to inspect requests and responses. It is highly recommended that you use a browser addon, such as Advanced Rest Client or RESTED, or use cURL to script your own tests. You could also write your own client-side tester using HTML, CSS, and JS!

A) Error Response

Implement a PHP function that returns a HTTP status code along with a custom message in the header and body of the response to the client. This function will be needed for all subsequent sections and should be used when an error occurs (invalid input, wrong request type etc.). The function must set the appropriate HTTP header **AND** provide the message in the response body in JSON format.

A good function definition would look like: `send_error($code, $message)`

The function should take 2 input parameters:

(Integer)	`$code`	will be the HTTP status code to return.
(String)	`$message`	will be a custom message text

The `code` variable contains a value which is also key in the `$responses` array provided. This array maps integer codes with their appropriate reason text. **Make sure to use this array!** You can get the server protocol from the `$_SERVER` superglobal.

The header should be of the form:

```
$PROTOCOL $CODE - $REASON
```

The response body should contain a valid JSON object of the form:

```
{
    "error": "$CODE - $REASON: $MESSAGE"
}
```

Where:

$PROTOCOL	=	HTTP Protocol used (*e.g.* HTTP/1.1)
$CODE	=	HTTP Status Code (*e.g.* 400)
$REASON	=	HTTP Status Code Reason (*e.g.* "Bad Request")
$MESSAGE	=	Custom message (*e.g.* "Age must be between 13-130")

Example Header:

```
HTTP/1.1 400 - Bad Request: Age must be between 13-130
```

Example Body:

```
{
    "error": 400 - Bad Request: Age must be between 13-130"
}
```

NOTE: You cannot use the `http_response_code()` function in PHP for this as it does not allow for custom reason messages.

B) Request Validation

Requests must be validated to ensure they are correct for the service. This means the request must be of the right type and must contain valid data.

All incoming requests must meet the following criteria:

1) The request **must** be POST: No other request types are accepted.
2) The request **must** contain POST data: An empty body is invalid
3) All required fields **must** be present in the POST data: `name`, `age`, and `email`
    a. Note the POST data may also include an **optional** field: `phone`

If **any** of the criteria above is not met, an appropriate HTTP status code must be returned to the client. The body of the response must also contain the HTTP status code and reason, along with a custom error message. You must use your error response function from **Part A**.

C) Data Validation

All user data must be validated on the server-side, to ensure it is safe, complete, and correct. Even though client-side validation *may* have been conducted, there is no guarantee that client-side code has not been modified, or even executed at all.

All incoming POST data for the service must be validated to meet the conditions below.

POST Field	Description	Validation Requirements
`name`	Name of the user	• MUST be between 2 and 100 characters long • MUST only contain characters a-z (upper and lower case), - (hyphen), or ' (apostrophe)
`age`	Age in years	• MUST be an integer value between 13-130
`email`	Email address	• MUST be validated using either: ○ The provided Regular Expression (See last page) ○ An appropriate regex found online (must be credited with the URL at which you found it)
phone	Phone number (optional)	• OPTIONAL: This field may be empty • MUST be exactly 10 characters long • MUST only contain digits (no letters or symbols) • MUST start with '04'

If any of these validation requirements are not met, an appropriate error message should be returned, using your error response function from **Part A**. You may return the first error encountered, all present errors, or some other variation.

D) User ID

If all validation is successful, your service should return success response, and a randomly generated user ID. The response should contain a JSON object with the format:

```
{
    "user_id": "some user id here"
}
```

The exact format of the user ID is up to you, but it must be randomly generated on the server.

E) Database

In a typical server-side application, once user data is validated it would then be written to a database. We will simulate a database by writing each successful request to a text file. For each request that passes validation, all received data should be written to file, in JSON format.

You **must** implement your "database" using **PHP classes**. An empty class file `Database.php` has been provided in the `class/` directory.

You **must** have a class property for each of the received fields: `name`, `age`, `email`, `phone`. Your class **must** implement the `JsonSerializable` interface and use this to write to the database file. This does mean that during the execution of your application, you will convert data from JSON (received from the client) to a `Database` object, then back to JSON for writing.

Note that there is **no requirement** to be able to programmatically read data back from your file, you only need to append the new data to the end.

F) Resources

Regular Expression Hints

`^`	:	Matches the start of a string
`$`	:	Matches the end of a string
`[]`	:	Matches all characters and character ranges contained within
`\w`	:	Matches a "word" character, same as `[A-Za-z0-9_]`
`*`	:	Matches the previous selector 0 or more times
`+`	:	Matches the previous selector 1 or more times
`?`	:	Matches the previous selector 0 or 1 times ("optionally matches")

Simple Email Validation Regular Expression

```
/^[a-zA-Z-]([\w-.]+)?@([\w-]+\.)+[\w]+$/
```

RESTED Browser Addon

https://addons.mozilla.org/en-US/firefox/addon/rested
https://chrome.google.com/webstore/detail/rested/eelcnbccaccipfolokglfhhmapdchbfg

HTTP Status Codes

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Links - PHP Documentation

Arrays:	https://secure.php.net/manual/en/book.array.php
Functions	https://secure.php.net/manual/en/language.functions.php
Classes & Objects	https://secure.php.net/manual/en/language.oop5.php
Superglobals:	https://secure.php.net/manual/en/language.variables.superglobals.php
$_SERVER:	https://secure.php.net/manual/en/reserved.variables.server.php

isset()	https://secure.php.net/manual/en/function.isset.php
array_push()	https://secure.php.net/manual/en/function.array-push.php
json_encode()	https://secure.php.net/manual/en/function.json-encode.php
header()	https://secure.php.net/manual/en/function.header.php
file_put_contents()	https://secure.php.net/manual/en/function.file-put-contents.php

JsonSerializable	https://secure.php.net/manual/en/jsonserializable.jsonserialize.php