Building a no-frills PHP CRUD App with Routing from Scratch

-Part 1-

Table of Contents

1 Preamble

This tutorial was written by **Tania Rascia** and adapted for this module by **Robert Smith**.

A complete walkthrough to setting up a PHP application with a MySQL database using modern, secure practices.

In this tutorial, we're going to learn how to make the beginnings of a very simple database app, using PHP and MySQL. It will be a **CRUD** application, which stands for **C**reate, **R**ead, **U**pdate, **D**elete.

A quick example of a CRUD application would be a database of employees for a company. From the control panel, an admin would be able to add a new employee (create), view a list of employees (read), change an employee's salary (update) or remove a fired employee from the system (delete). In this lesson, we'll only be **creating** and **reading** entries. Update and Delete are covered in part two.

CRUD task	SQL Query	HTTP request method
Create	INSERT	PUT
Read	SELECT	GET
Update	UPDATE	PUT
Delete	DELETE	DELETE

Searching the internet for *how to connect to MySQL with PHP* will lead to a lot of outdated resources using **deprecated** code. For this reason, it is important that you follow these instructions (and those on SymfonyCasts). My aim here, is to create a very simple walkthrough that will leave you with a technically functioning app that uses modern and secure methods.

That being said, I leave you with the disclaimer that this tutorial is meant for **beginner learning purposes** - there can always be improvements to security and maintainability for code in the real world, e.g. **OOPHP** and **PHP frameworks** such as **Symfony** and **Laravel**.

1.1 Prerequisites

- **A basic knowledge of HTML** - We'll be using **HTML forms**.
- **A basic knowledge of PHP** – See SymfonyCasts PHP course 1
- **A local PHP and MySQL environment** – Laragon, LAMP, XXAMP etc…
- **A database management app** - You can use HeidiSQL, PHPMyAdmin, SQL Workbench, **Sequel Pro** on a Mac, or **SQLYog** on Windows etc.... Use any graphical user interfaces (GUI) to view the content of your database.
    - Of course, you may also use the CLI

1.2 Goals

- Connect to a MySQL database with PHP using the **PDO (PHP Data Objects)** method.
- Create an installer script that creates a new database and a new table with structure.
- Add entries to a database using a HTML form and **prepared statements**.
- Filter database entries and print them to the HTML document in a table.

2 Building the front end

To start, we have a PHP localhost set up, as mentioned in our prerequisites. Let's create a directory called **public/** in the root of our project. This is where I'm going to put all my client-facing code, or what would be pages accessible from the internet.

It is possible to setup localhost domains, such as mysite.test or DB.dev, on Laragon. This effectively maps to 'Localhost'. We will just use Localhost here. If you see DB.dev in an example, just replace it with Localhost.

Please make sure you read the above note before continuing. We don't have a database set up yet, before we do that we're just going to set up the HTML front end in order to be prepared to start interacting with that data.

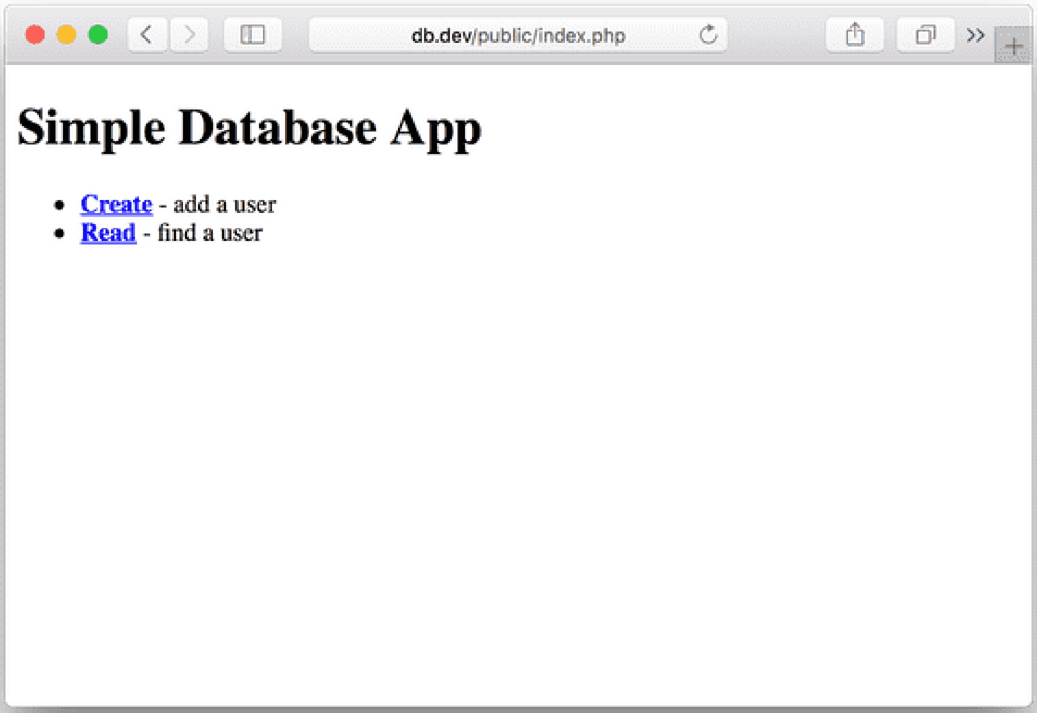
2.1 Index Page and Templating

Our main/home page will be located at **index.php**, so create that file in your **public/** directory.

public/index.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <title>Simple Database App</title>

    <link rel="stylesheet" href="css/style.css" />
  </head>

  <body>
    <h1>Simple Database App</h1>

    <ul>
      <li>
        <a href="create.php"><strong>Create</strong></a> - add a user
      </li>
      <li>
        <a href="read.php"><strong>Read</strong></a> - find a user
      </li>
    </ul>
  </body>
</html>
```

Right now, all we have is a basic HTML skeleton that will link to our **create** and **read** pages. Here's what it looks like:



Simple Database App

- Create - add a user
- Read - find a user

Since we want to keep our application relatively **DRY (don't repeat yourself)**, we're going to divide our page into layout sections.

Create a **templates/** directory in public, and make a **header.php** and **footer.php**. You'll take everything from the `<h1>` tag and up and put it in the header.

public/templates/header.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <title>Simple Database App</title>

    <link rel="stylesheet" href="css/style.css" />
  </head>

  <body>
    <h1>Simple Database App</h1>
```

And here's the footer.

```
                                                public/templates/footer.php
</body>
</html>
```

All that remains in **index.php** at this point are the links to our two other pages.

```
                                                            public/index.php
<ul>
  <li>
    <a href="create.php"><strong>Create</strong></a> - add a user
  </li>
  <li>
    <a href="read.php"><strong>Read</strong></a> - find a user
  </li>
</ul>
```

We want to include the header and footer code in all our front end pages, so we'll be using a **PHP include function** to pull that code in.

```
                                                            public/index.php
<?php include "templates/header.php"; ?>

<ul>
  <li>
    <a href="create.php"><strong>Create</strong></a> - add a user
  </li>
  <li>
    <a href="read.php"><strong>Read</strong></a> - find a user
  </li>
</ul>

<?php include "templates/footer.php"; ?>
```

Now the front end of our index file looks the same as before, but we have the reusable layout code that we can use in our other pages.

> *Note: newly added bits of code are yellow font*

2.2 Add a new user page

Now we're going to make a file called **create.php** back in our **public/** directory. This will be the page we use to add a new user to the database. We'll start the file with our header and footer loaded in.

```
                                                           public/create.php
<?php include "templates/header.php"; ?>
<?php include "templates/footer.php"; ?>
```

I'm going to create a simple form here that gathers the first name, last name, email address, age, and location of a new user.

```
public/create.php
<?php include "templates/header.php"; ?>

<h2>Add a user</h2>
    <form method="post">
      <label for="firstname">First Name</label>
      <input type="text" name="firstname" id="firstname" required>
      <label for="lastname">Last Name</label>
      <input type="text" name="lastname" id="lastname" required>
      <label for="email">Email Address</label>
      <input type="email" name="email" id="email" required>
      <label for="age">Age</label>
      <input type="text" name="age" id="age">
      <label for="location">Location</label>
      <input type="text" name="location" id="location">
      <input type="submit" name="submit" value="Submit">
    </form>

    <a href="index.php">Back to home</a>

    <?php include "templates/footer.php"; ?>
```

You'll notice that for each entry, I have a `<label>`, and each `<input>` has a `name` and `id` attribute.

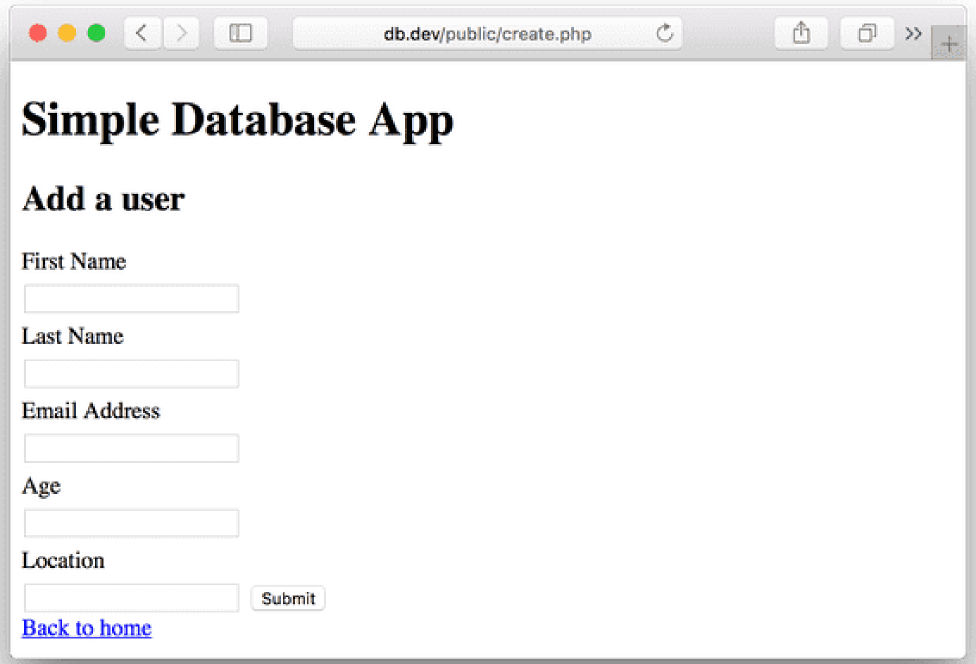
Whenever we create forms with HTML, we need to make sure they're **accessible for all users**, and we do that by creating labels and associating each one to a specific. An input is associated to a label by its id.

So why do I have `name="firstname"` as well as `id="firstname"`, if only the id is necessary to associate the input to the label?

The `name` attribute is how PHP identifies and utilizes the data of the input, which we'll start getting into further on in the article. Therefore both the `name` and `id` attributes are necessary, but for different reasons.

Before I display the front end of the **create.php** code, let's quickly create a **css/** folder and make **style.css**. CSS and style is not a focus of this tutorial, but I'm going to add a line of CSS code to make the forms easier to read.

```
public/css/style.css
label {
  display: block;
  margin: 5px 0;
}
```



Simple Database App

Add a user

First Name

Last Name

Email Address

Age

Location

Submit

Back to home

We have not specified a form action, so pressing the `submit` button will perform the action on the same page. Since we haven't written any PHP code to process the form yet, it won't do anything.

2.3 Query users page

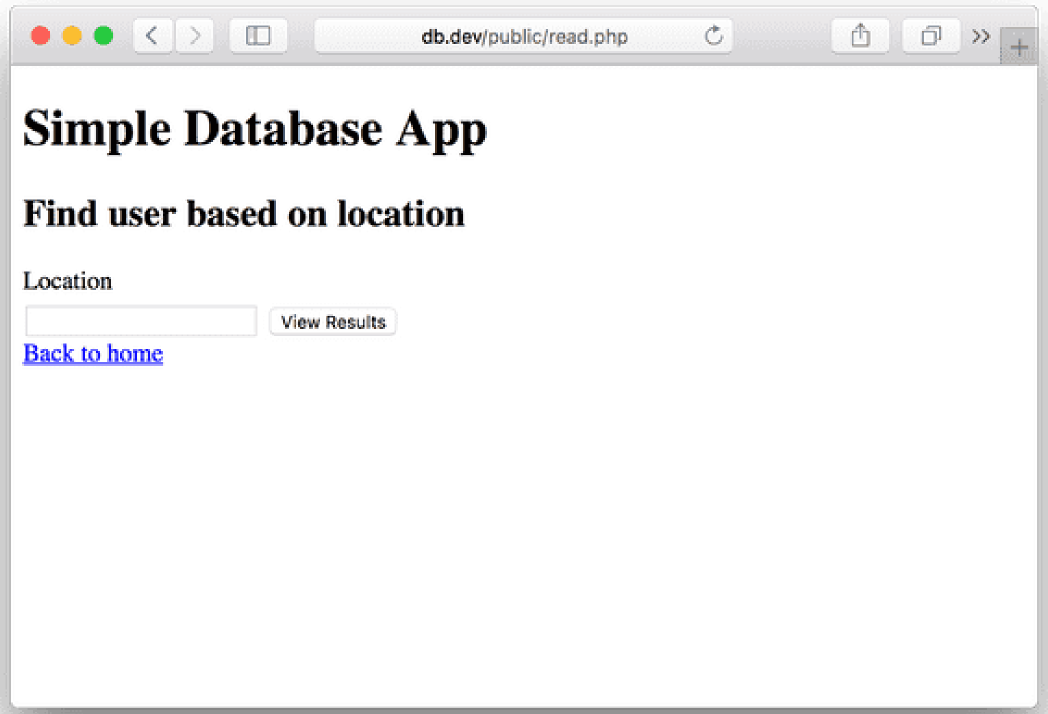
Finally, we're going to create our **read.php** file, which will query the list of users by a parameter (in this case, location) and print out the results.

Again, we'll start with the header and footer code.

public/read.php
```
<?php include "templates/header.php"; ?>
<?php include "templates/footer.php"; ?>
```

Then we'll add a small form for searching for users by location.

public/read.php
```
<?php include "templates/header.php"; ?>

    <h2>Find user based on location</h2>

    <form method="post">
      <label for="location">Location</label>
      <input type="text" id="location" name="location">
      <input type="submit" name="submit" value="View Results">
    </form>

    <a href="index.php">Back to home</a>

    <?php include "templates/footer.php"; ?>
```


db.dev/public/read.php

Simple Database App

Find user based on location

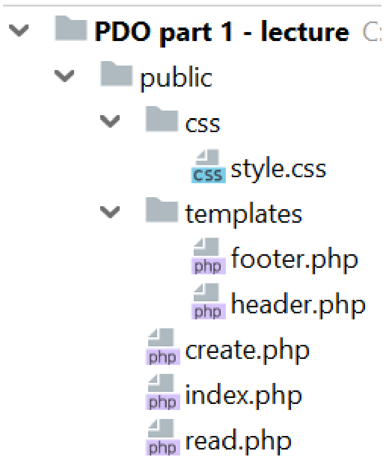
Location

View Results

Back to home

Now we have all the front-end code set up and we can start on the back end. Here's everything you should have so far (a more visual representation of that on the right).

```
Root_Dir
        |-- public/
                |-- css/
                    |-- style.css
                |-- templates/
                    |-- header.php
                    |-- footer.php
                |-- index.php
                |-- create.php
                |-- read.php
```

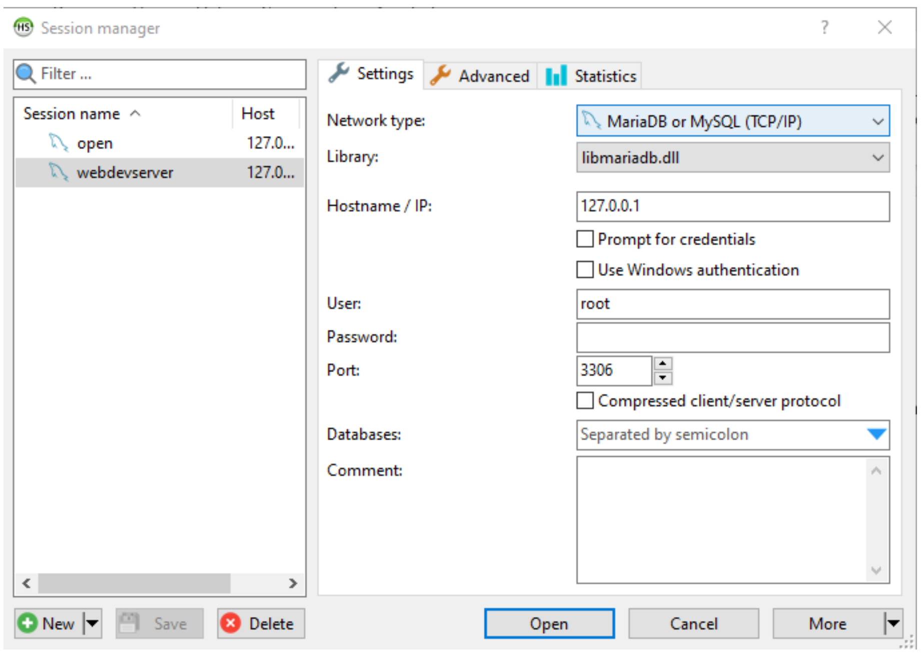


3 Initializing the database

Now that we have a front end, we're going to create the database. We could do this through the GUI of HeidiSQL or whatever database manager we're using, but I want to show you how to do it with actual SQL statements and PHP in an installer script (that is a script that will setup a new DB and/or Table).

First, let's get into the database. You can access HeidiSQL by clicking the 'Database' button at the bottom of the Laragon panel.

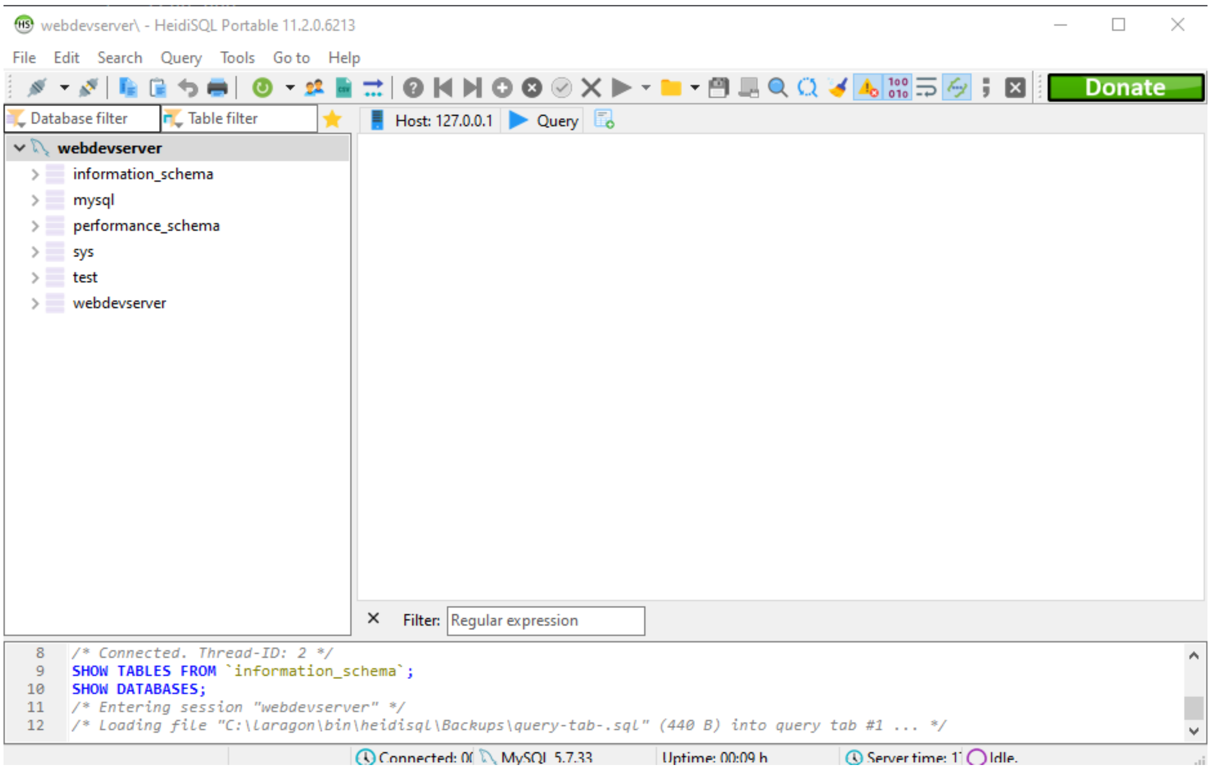
Here's the login page for the front end of our database.



Network type: MariaDB or MySQL (TCP/IP)
Library: libmariadb.dll
Hostname / IP: 127.0.0.1
☐ Prompt for credentials
☐ Use Windows authentication
User: root
Password:
Port: 3306
☐ Compressed client/server protocol
Databases: Separated by semicolon
Comment:

Your host will be `localhost` or `127.0.0.1`, which translate to the same thing for our purposes. Username and password will be `root` and blank by default. Entering that information in, you should be able to enter localhost.

> *If you've had a previous version of MySQL installed, you may have changed the default username (root) and password (blank). If this is the case than use those login credentials here.*



```
/* Connected. Thread-ID: 2 */
SHOW TABLES FROM `information_schema`;
SHOW DATABASES;
/* Entering session "webdevserver" */
/* Loading file "C:\laragon\bin\heidisql\Backups\query-tab-.sql" (440 B) into query tab #1 ... */
```

Moving away from HeidiSQl for now, create a directory called **data/** in the root, and create a text file called **init.sql**. This will be our database initializing code.

data/init.sql

```
CREATE DATABASE test;

  use test;

  CREATE TABLE users (
    id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    firstname VARCHAR(30) NOT NULL,
    lastname VARCHAR(30) NOT NULL,
    email VARCHAR(50) NOT NULL,
    age INT(3),
    location VARCHAR(50),
    date TIMESTAMP
  );
```

This is relatively straightforward SQL code, so even if you've never seen it before, it should be easy to understand. Here's what the above means in plain English:

We're going to create a database called `test`. Then we're going to make sure we're using `test` for the rest of our code. In the `test` database, we'll create a table called `users` with 7 fields inside - **id**, **firstname**, **lastname**, **email**, **age**, **location**, and **date**. Next to each field is more information, options, and settings for each.

`INT()` - this is an **Integer**. We specified `INT(11)`, which means up to 11 characters

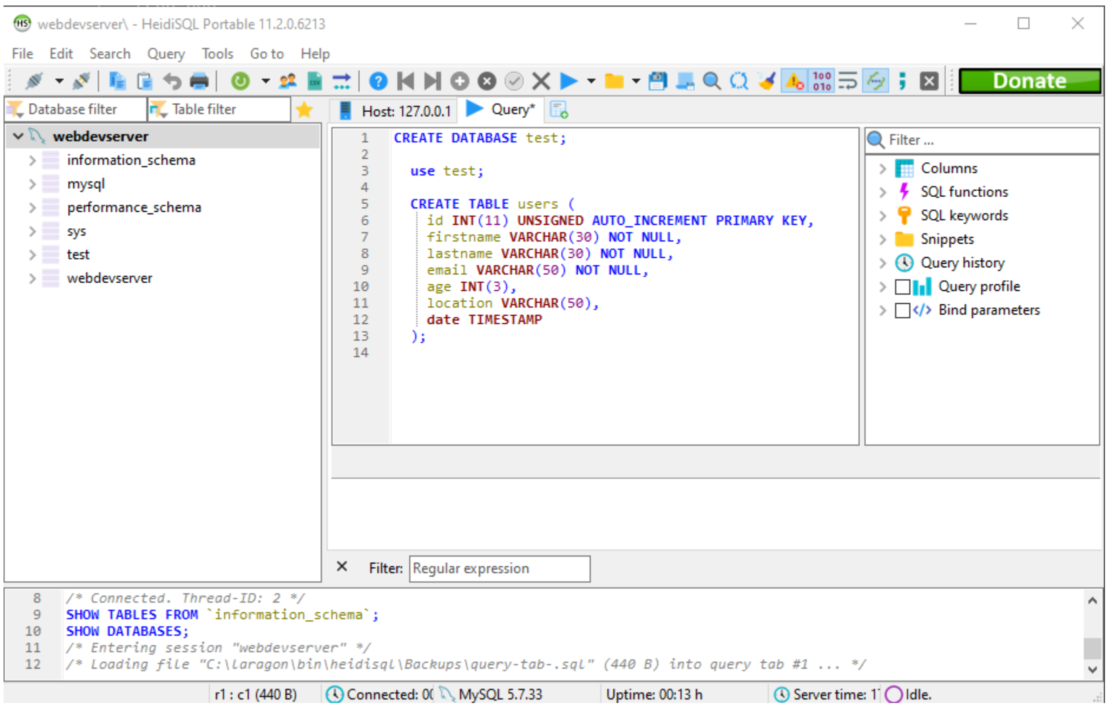
`AUTO_INCREMENT` - this is a number that will automatically increase with each entry.

`VARCHAR()` - meaning **Variable Character**, this is a string that can contain letters and numbers. The number inside is the max amount of characters allowed.

`TIMESTAMP` - this will add the current time in `YYYY-MM-DD HH:MI:SS` format by default.

3.1 Testing the SQL query

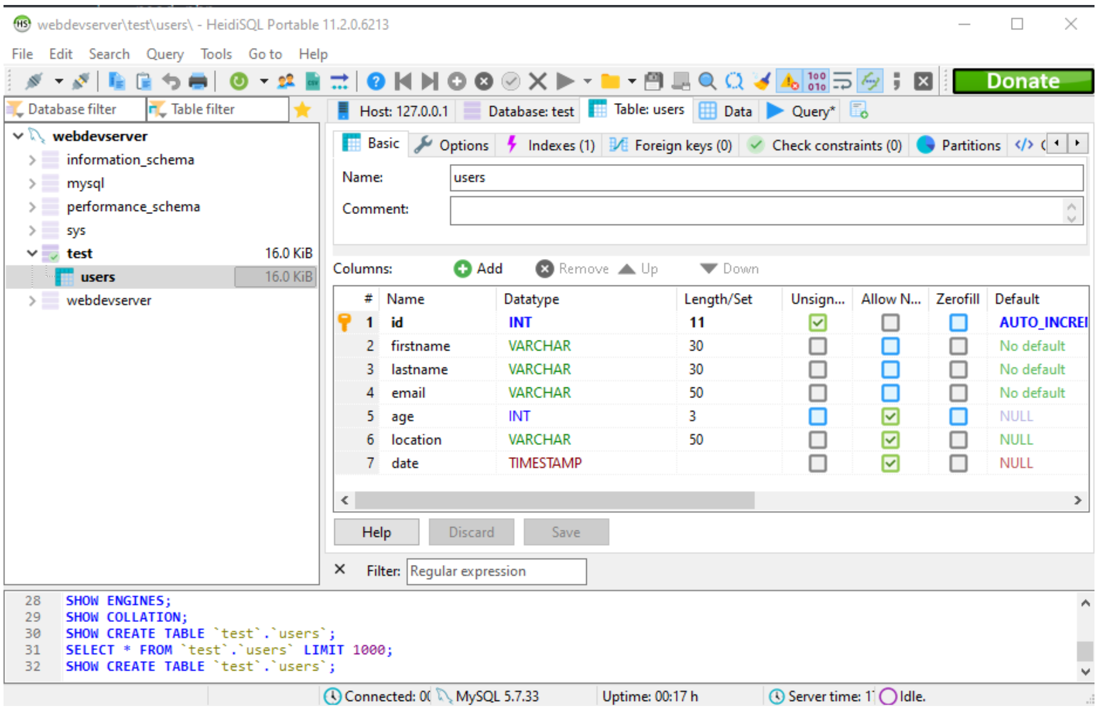
If you really want to see what this code will do before we create the installer, you can paste it into the **Query** panel of your database program, highlight the text and **Run Selection**.



```
CREATE DATABASE test;

  use test;

  CREATE TABLE users (
    id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    firstname VARCHAR(30) NOT NULL,
    lastname VARCHAR(30) NOT NULL,
    email VARCHAR(50) NOT NULL,
    age INT(3),
    location VARCHAR(50),
    date TIMESTAMP
  );
```

```
/* Connected. Thread-ID: 2 */
SHOW TABLES FROM `information_schema`;
SHOW DATABASES;
/* Entering session "webdevserver" */
/* Loading file "C:\laragon\bin\heidisql\Backups\query-tab-.sql" (440 B) into query tab #1 ... */
```

If you did that, you'll see I now have a `test` database, a `users` table, and all the database structure (the table is empty and that's fine for now).



Name: users

Comment:

Columns:

#	Name	Datatype	Length/Set	Unsign...	Allow N...	Zerofill	Default
1	id	INT	11	✓	☐	☐	AUTO_INCREI
2	firstname	VARCHAR	30	☐	☐	☐	No default
3	lastname	VARCHAR	30	☐	☐	☐	No default
4	email	VARCHAR	50	☐	☐	☐	No default
5	age	INT	3	☐	✓	☐	NULL
6	location	VARCHAR	50	☐	✓	☐	NULL
7	date	TIMESTAMP		☐	✓	☐	NULL

```
SHOW ENGINES;
SHOW COLLATION;
SHOW CREATE TABLE `test`.`users`;
SELECT * FROM `test`.`users` LIMIT 1000;
SHOW CREATE TABLE `test`.`users`;
```

So we know our SQL works properly and has no errors. If you did that, **delete/drop the database now** because we're going to start over and do it through PHP script. To do that, right click the DB `test` in the left panel and choose `Drop`.

3.2 Using PDO to connect to a database

We're going to use **PDO (PHP Data Objects)** to connect to the database. The other major option is **MySQLi**. The critical difference between the two is that you can use PDO to connect to any number of **database vendors**, while mysqli code will only work with MySQL. Although we're using a MySQL database, PDO is more extendable in the future, and generally the preferred choice for new projects. So let's create that connection.

Create a file called **install.php** in the root of your directory.

We'll create a new PDO() object and place it into a variable named $connection.

```
install.php
<?php $connection = new PDO(); ?>
```

The PDO object will ask for four parameters:

- **DSN (data source name)**, which includes type of database, host name, and database name (optional)
- Username to connect to host
- Password to connect to host
- Additional options

$connection = new PDO(**DSN, username, password, options** *);*

Here's how that ends up looking after we fill in all the parameters.

```
install.php
$connection = new PDO("mysql:host=localhost", "root", "root",
  array(
      PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
  );
);
```

That's looking good. Though it's never a good idea to hardcode values like this. Let's organize it a bit more by putting all our database information into variables. Let's replace the values with variables.

These variables don't exist yet. We'll create those next.

```
install.php
$connection = new PDO("mysql:host=$host", $username, $password, $options);
```

We'll create a **config.php** file that contains all the variables we can refer from.

Setting values for our variables in another file or class provides an extra layer between the public and our sensitive data. It is a more secure approach.

The ONLY purpose of the config file is to store the DB login data.

Note: It is also possible to store our values in an associative array as is the case in the SymfonyCast tutorials.

```php
                                                               config.php
<?php

/**
  * Configuration for database connection
  *
  */

$host         = "localhost";
$username     = "root";
$password     = "";
$dbname       = "test"; // will use later
$dsn          = "mysql:host=$host;dbname=$dbname"; // will use later
$options      = array(
                  PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
                );
```

Here's what we have in our installer so far. We're pulling in the database variables via **config.php** using `require`, which is similar to an `include`, except we're explicitly stating that the file is necessary for the script to run.

PHP script will load with warning is there is an issue with `include`. `require` is different because it stops the PHP from loading completely, if there is a problem with the required script.

As such, the `install.php` script will never run if there is a problem with the `config.php` script.

```php
                                                              install.php
require "config.php";

$connection = new PDO("mysql:host=$host", $username, $password, $options);
```

Now it's time to use SQL code we created at the start of this section. We'll be placing the contents of the **data/init.sql** file into a variable using the `file_get_contents()` function, and executing it with the `exec()` function.

```php
                                                              install.php
require "config.php";

$connection = new PDO("mysql:host=$host", $username, $password, $options);

$sql = file_get_contents("data/init.sql");
$connection->exec($sql);
Echo "DB setup";
```

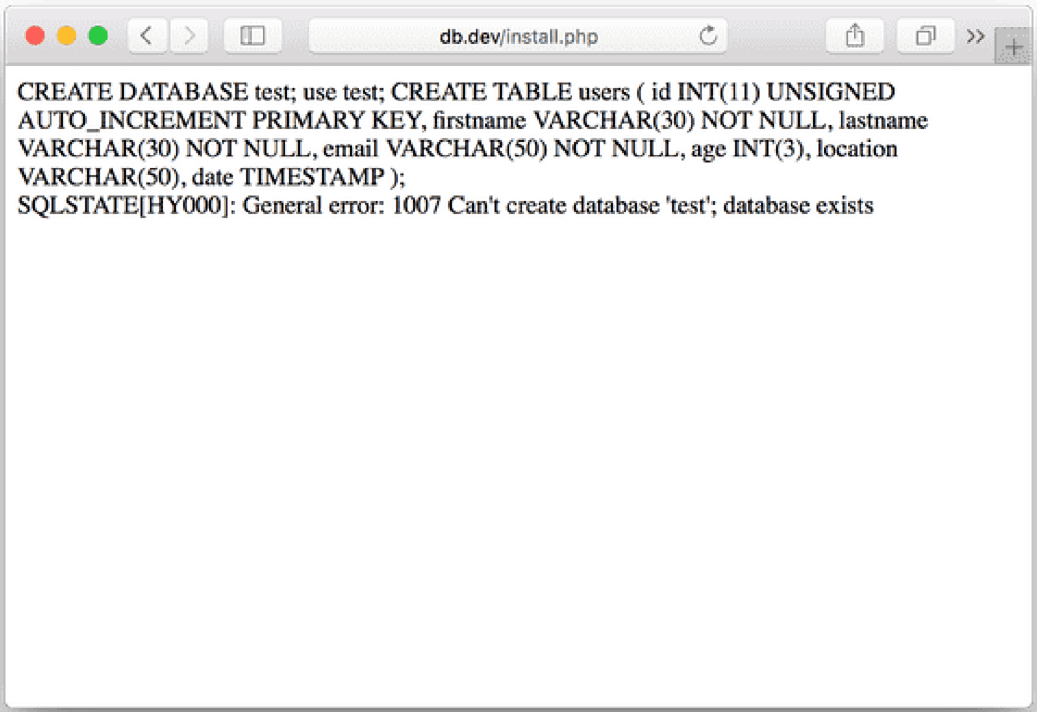
At this point, we're going to want to use **Exceptions** to attempt to run the script and catch errors. We'll do this by putting all our code in a `try/catch` block, which looks like this:

```
try {
// code to execute
} catch() {
// exception
}
```

Let's put our database code in the `try` block and show our `PDOException` error message if something goes wrong trying to set up the database. Here's the final code for the installer.

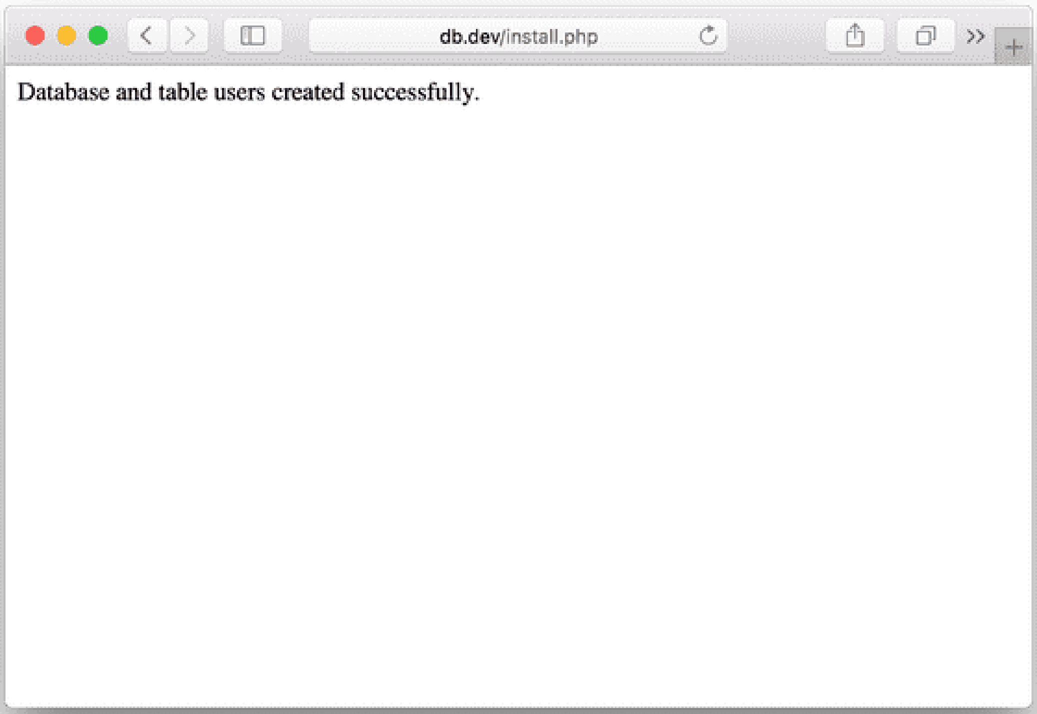
```
                                                                  install.php
<?php

/**
  * Open a connection via PDO to create a new database and table with
structure.   */

require "config.php";

try {
  $connection = new PDO("mysql:host=$host", $username, $password,
$options);
  $sql = file_get_contents("data/init.sql");
  $connection->exec($sql);

  echo "Database and table users created successfully.";
} catch(PDOException $error) {
  echo $sql . "<br>" . $error->getMessage();
}
```

To run the install, just navigate to your **install.php** file on the front end (browser), as illustrated below:



```
CREATE DATABASE test; use test; CREATE TABLE users ( id INT(11) UNSIGNED
AUTO_INCREMENT PRIMARY KEY, firstname VARCHAR(30) NOT NULL, lastname
VARCHAR(30) NOT NULL, email VARCHAR(50) NOT NULL, age INT(3), location
VARCHAR(50), date TIMESTAMP );
SQLSTATE[HY000]: General error: 1007 Can't create database 'test'; database exists
```

This screenshot depicts an error which states that the database already exists. If you didn't delete your database from our test before, you will get this error, in which case you'll need to drop the database for the installer to run.


db.dev/install.php

```
Database and table users created successfully.
```

This is more like it. Congratulations, you just made an installer script to set up a new database and table with structure!

So what did we do in this section? (1). We created a connection to the DB. (2). We put our sensitive login details in a secure location. (3). We executed the SQL queries saved in `init.sql`*.*

In this exercise we set up a new DB and Table… but these are just SQL queries, and every time we want to query a DB from PHP we follow the same steps. The queries may be different, but the process is the same, i.e., make a connection to the DB > then execute the query…. All the while, make things as secure as possible.

Therefore, you should see the benefits of separating the tasks here. If we put the DB connection code into its own class/file we can call it when we want to run a query… instead of retyping the code each time. Remember, its bad to repeat code (DRY).

4 Adding a new user (Create)

OK, we have a database, and we have some HTML forms. Now we're going to write some code to make those forms do something.

In PHP, when you submit a form, all the inputs are placed into a `$_POST` or `$_GET` superglobal array. Because we are using sensitive user data in this example, we will use `POST` as it is the more secure method.

So when the server does a POST request, it stores the form data in the $_POST array, which if you remember, is an associative array with key/value pairs. Therefore, the value (entered by the user) from the form field `<input type="text" name="firstname">` will be accessible from the $_POST array by referencing the array key i.e., `$_POST['firstname']`.

We're going to go back to the **public/create.php** file we created earlier. Right now, it's just a form with a header and footer being pulled in. The new code we write will be added to the top of the file.

First, we're going to tell this code only to run if the form has been submitted.

```
if (isset($_POST['submit'])) {}
```

Just like with the installer, we're going to `require` the config file, and use a `try/catch` Exception to connect to the database.

If you'll notice, in the **install.php** script I was only connecting to `mysql:host=$host` in the first parameter (DSN). I didn't specify a database name because we were creating the database in the file. Now that the database (named `test`) exists, I'm adding that to the first parameter. `$dsn` is set to `mysql:host=$host;dbname=$dbname`.

```
if (isset($_POST['submit'])) {
  require "../config.php";

  try {
    $connection = new PDO($dsn, $username, $password, $options);
    // insert new user code will go here

  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }
}
```

Just after the PDO connection, lets replace the comment with a new array that will hold all our submitted form values.

```
$new_user = array(
  "firstname" => $_POST['firstname'],
  "lastname"  => $_POST['lastname'],
  "email"     => $_POST['email'],
  "age"       => $_POST['age'],
  "location"  => $_POST['location']
);
```

We should really be sanitizing input from the $_POST array here. For simplicity's sake, lets come back to that later.

Now, the SQL code we want to execute will look like this:

```
INSERT INTO tablename (Column/s) VALUES (value/s);
```

In our specific case, it will be the below code:

```
        INSERT INTO users (firstname, lastname, email, age, location)
        VALUES(:firstname, :lastname, :email, :age, :location)
```

Of course, the line above won't work because it's not PHP code… it's just a string. Remember, the data we want to add here comes from the array we just created called `$new_user`… and in this case we want to refer to the key name rather than the data. Here's the code we need:

```
        $sql = "INSERT INTO users (" . implode(', ', array_keys($new_user))
        .") values (:". implode(', :', array_keys($new_user)).")";
```

We could write out that code by hand and add each value every time we add one to add, but then we're updating things in three places and it becomes messy. The line of code above might look a little complex but it's quite simple. I'll break it down for you:

If we use `var_dump()` to see the contents of `array_keys($new_user),` we can see that this method returns an array listing all of our array keys (these are also our column headings in the 'user' table). We pass that array of array-keys to the `implode()` function. The implode function's job is to return a string. So the data was an array going into the implode function but a string coming out of the function. Also… note that the implode function will accept another argument which will delimit each array item.

In this case the delimiter is comma-space-colon:

```
        implode(', :', array_keys($new_user))
```

Using the implode functions will output the exact same string as typing out all of the query elements manually. Though if you find it hard to get your head around, feel free to type the string manually.

Add the code below just after the `$new_user` array.

```
$sql = sprintf( "INSERT INTO %s (%s) values (%s)", "users", implode(", ",
array_keys($new_user)), ":" . implode(", :", array_keys($new_user)) );
```

*Note: this code puts the query into a function called **`sprintf()`**, which writes a formatted string to a variable.*

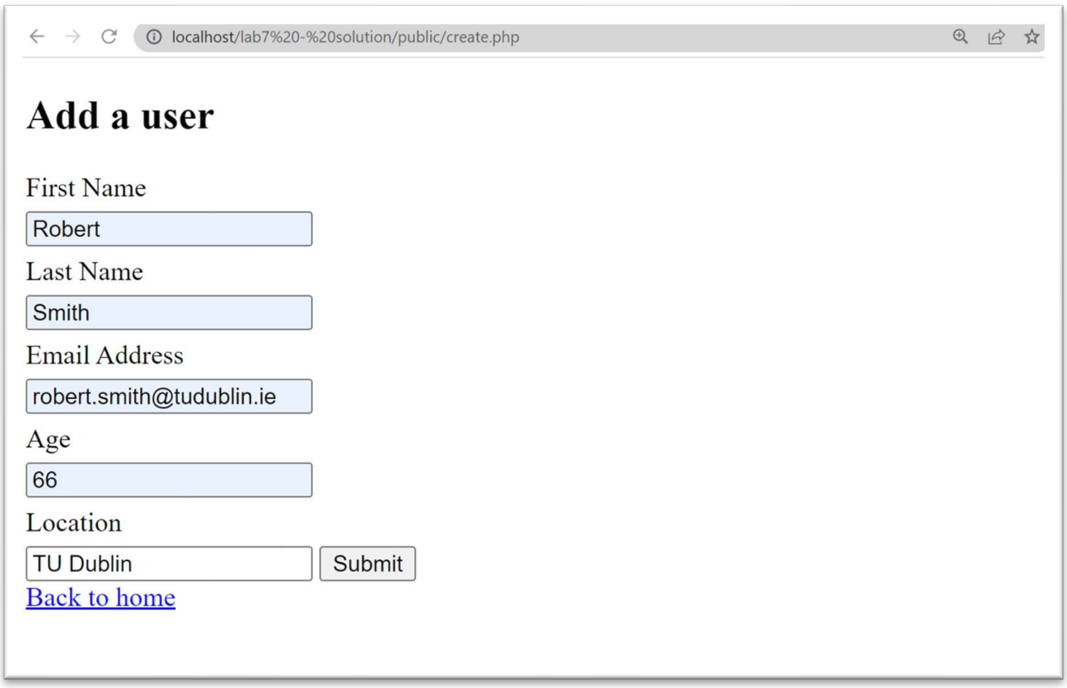
Now we'll just `prepare` and `execute` the code. Add these lines of code just after we define `$sql`

```
$sql = sprintf( "INSERT INTO %s (%s) values (%s)", "users", implode(", ",
array_keys($new_user)), ":" . implode(", :", array_keys($new_user)) );

$statement = $connection->prepare($sql);
$statement->execute($new_user);
```

Here is the full code inside our **create.php** file (this is not the final code, we've more work to do).

```
                                                                     public/create.php
<?php include "templates/header.php"; ?><h2>Add a user</h2>

<?php
if (isset($_POST['submit'])) {
  require "../config.php";

  try {
    $connection = new PDO($dsn, $username, $password, $options);
    $new_user = array(
      "firstname" => $_POST['firstname'],
      "lastname"  => $_POST['lastname'],
      "email"     => $_POST['email'],
      "age"       => $_POST['age'],
      "location"  => $_POST['location']
      );

     $sql = sprintf("INSERT INTO %s (%s) values (%s)", "users",
      implode(", ", array_keys($new_user)),
      ":" . implode(", :", array_keys($new_user)));

      $statement = $connection->prepare($sql);
      $statement->execute($new_user);


  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }

}
?>
    <form method="post">
      <label for="firstname">First Name</label>
      <input type="text" name="firstname" id="firstname" required>
      <label for="lastname">Last Name</label>
      <input type="text" name="lastname" id="lastname" required>
      <label for="email">Email Address</label>
      <input type="email" name="email" id="email" required>
      <label for="age">Age</label>
      <input type="text" name="age" id="age">
      <label for="location">Location</label>
      <input type="text" name="location" id="location">
      <input type="submit" name="submit" value="Submit">
    </form>

    <a href="index.php">Back to home</a>

    <?php include "templates/footer.php"; ?>
```

Now the form is all ready to send. I'm going to fill out my information and submit it.


localhost/lab7%20-%20solution/public/create.php

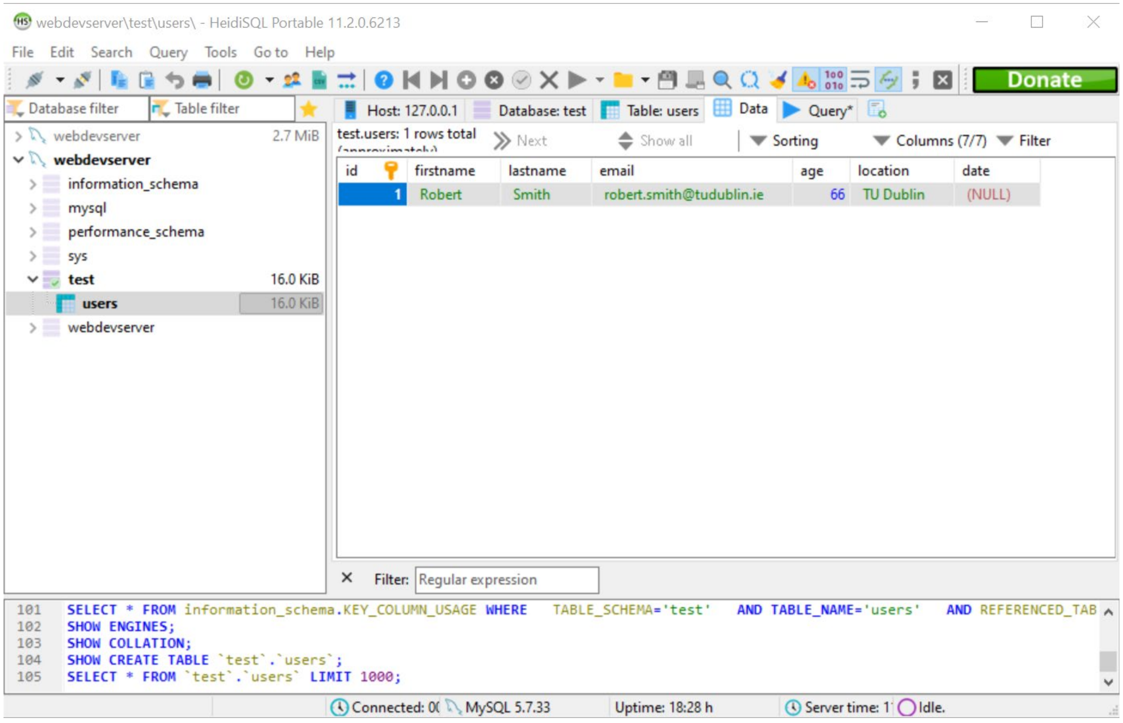
Add a user

First Name	Robert
Last Name	Smith
Email Address	robert.smith@tudublin.ie
Age	66
Location	TU Dublin

Submit

Back to home

Going into my MySQL, if all went well, I can now view my entry in the database!



test.users: 1 rows total

id	firstname	lastname	email	age	location	date
1	Robert	Smith	robert.smith@tudublin.ie	66	TU Dublin	(NULL)

```
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE   TABLE_SCHEMA='test'   AND TABLE_NAME='users'   AND REFERENCED_TAB
SHOW ENGINES;
SHOW COLLATION;
SHOW CREATE TABLE `test`.`users`;
SELECT * FROM `test`.`users` LIMIT 1000;
```

Connected: 0(MySQL 5.7.33 Uptime: 18:28 h Server time: 1 Idle.

Excellent! There's just one more thing I want to do. Right now, after submitting the form, there's nothing letting me know that the user was submitted successfully.

Were almost done with this task. Just some housekeeping to do.

4.1 Modularising the DB connection script

I mentioned in the previous section that we should use a single script for the DB connection and call that script every time we need it. This helps us avoid repeating lines of code, is more efficient, and reduces potential points of error.

Create a new directory called **src** and a new script called **src/DBconnect.php**.

```
                                                        src/DBconnect.php
<?php

require_once '../config.php'; //access the login values

try {
     $connection = new PDO($dsn, $username, $password, $options);
       echo 'DB connected';
} catch (\PDOException $e) {
     throw new \PDOException($e->getMessage(), (int)$e->getCode());
}

?>
```

//new PDO() returns an object that can be used to interact with the database. Here we store the returned PDO object into a variable called $connection

This code could be added to a DB connection class to make it more OOPHP friendly.

Now **DBconnect.php** can be used every time we connect to the DB. Edit the `if` statement in **create.php** so that it now uses **DBconnect.php** to connect to the DB.

```
                                                       public/create.php
…
if (isset($_POST['submit'])) {
  ~~require "../config.php";~~

  try {
    ~~$connection = new PDO($dsn, $username, $password, $options);~~
      require_once '../src/DBconnect.php';


  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }

}
…
```

There is no longer a need to require config.php because this is done in DBconnect.php.

The line that creates the new DB connection is replaced with a call to the code in DBconnect.php

Now every time we need to connect to the DB, we will do it this way.

4.2 Form Sanitisation

Since in this case we're going to print out a `$_POST` variable to the HTML, we need to properly sanitise the user input to aid in preventing XSS attacks.

Create a new file called **common.php** in the root directory of your project.

Again, this is a good place to create a class for OOPHP

```
                                                                   common.php
<?php

function escape($data) {
  $data = htmlspecialchars($data, ENT_QUOTES | ENT_SUBSTITUTE, "UTF-8");
  $data = trim($data);
  $data = stripslashes($data);
      return ($data);
}
```

With this function, we can wrap any variable in the `escape()` function, and the HTML entities will be protected.

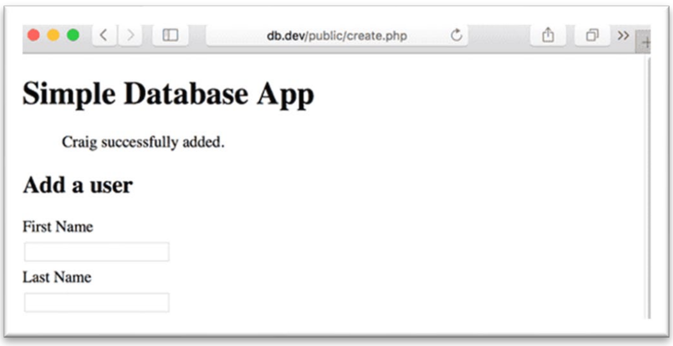
Back in **public/create.php**, add a `require "common.php";` and edit the $new_user[] array so that the escape function is applied to all input.

```
  $new_user = array(
            "firstname" => escape($_POST['firstname']),
            "lastname"  => escape($_POST['lastname']),
            "email"     => escape($_POST['email']),
            "age"       => escape($_POST['age']),
            "location"  => escape($_POST['location'])
        );
```

Now I'm just going to add the below `if` statement just above the HTML form. This `if` statement will check to see if a `$_POST` was submitted, and if a `$statement` was successful. If so, it will print a success message that includes the first name of the successfully added user.

*Its been a little while ago now, and we've covered a lot since then… but remember `$statement` is where we prepared and executed the SQL query in **create.php**.*

```
…
if (isset($_POST['submit']) && $statement)
    {
    echo $new_user['firstname']. ' successfully added';
    }
?>…
```



Simple Database App

Craig successfully added.

Add a user

First Name

Last Name

And that's everything! Here's the final code to add a new user.

```
                                                                public/create.php
<?php

if (isset($_POST['submit'])) {
    require "../common.php";

    try {
        require_once '../src/DBconnect.php';

        $new_user = array(
            "firstname" => escape($_POST['firstname']),
            "lastname"  => escape($_POST['lastname']),
            "email"     => escape($_POST['email']),
            "age"       => escape($_POST['age']),
            "location"  => escape($_POST['location'])
        );
        $sql = sprintf(
            "INSERT INTO %s (%s) values (%s)",
            "users",
            implode(", ", array_keys($new_user)),
            ":" . implode(", :", array_keys($new_user))
        );
        $statement = $connection->prepare($sql);
        $statement->execute($new_user);

    } catch(PDOException $error) {
        echo $sql . "<br>" . $error->getMessage();
    }
}
require "templates/header.php";
if (isset($_POST['submit']) && $statement){
    echo $new_user['firstname']. ' successfully added';
    }
?>

<h2>Add a user</h2>
<form method="post">
    <label for="firstname">First Name</label>
    <input type="text" name="firstname" id="firstname">
    <label for="lastname">Last Name</label>
    <input type="text" name="lastname" id="lastname">
    <label for="email">Email Address</label>
    <input type="text" name="email" id="email">
    <label for="age">Age</label>
    <input type="text" name="age" id="age">
    <label for="location">Location</label>
    <input type="text" name="location" id="location">
    <input type="submit" name="submit" value="Submit">
</form>
<a href="index.php">Back to home</a>
<?php include "templates/footer.php"; ?>
```

5 Viewing and filtering users (Read)

Now that we’ve looked at “creating” items in the DB, lets now turn our attention to the "read" aspect of our CRUD app. We already created the front end in **public/read.php**.

Really quickly, let's add a small amount of CSS to our **public/css/style.css** file to make the tables legible once we create them.

```
                                                   public/css/style.css
table {
  border-collapse: collapse;
  border-spacing: 0;
}

td,
th {
  padding: 5px;
  border-bottom: 1px solid #aaa;
}
```

Now we're going to use the same `requires` from our **create.php** page, as well as the `try/catch` block for connecting to the database. Add this code before `include "../templates/header.php”;`.

```
                                                        public/read.php
<?php
if (isset($_POST['submit'])) {
  try {
    require "../common.php";

    require_once '../src/DBconnect.php';

        // SQL query code will go here
  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }
}
?>
```

Now that we have established a DB connection for the **read.php** page (a new connection must be established each time), we'll write a `SELECT` SQL query. We're going to select all (`*`) from the users table, and filter by location. The first step in this process is to store the query string in a variable called `$sql`.

```
$sql = "SELECT *
  FROM users
  WHERE location = :location";
```

Then we'll grab the location value from the `$_POST` array and store that in the variable `$location`.

```
$location = $_POST['location'];
```

After that, we `prepare`, `bind`, and `execute` the SQL query.

```
$statement = $connection->prepare($sql);
$statement->bindParam(':location', $location, PDO::PARAM_STR);
$statement->execute();
```

`prepare`, `bind`, and `execute` refer to existing functions used for SQL query statements in PHP. These are used to create "prepared statements".

*`prepare()` - **PDO::prepare** — Prepares a statement for execution and returns a PDO statement object.*

*`bindParam()` - **PDOStatement::bindParam** — Binds a parameter to the specified variable name*

*`execute()` - **PDOStatement::execute** — Executes a prepared statement*

Note that `execute()` is not the same as `exec()` which was used in install.php. `execute()` is for prepared statements only.

*`Exec()` - **PDO::exec** — Execute an SQL statement and return the number of affected rows*

Finally, we'll fetch the result.

```
$result = $statement->fetchAll();
```

*fetchAll() - **PDOStatement::fetchAll** — Fetches the remaining rows from a result set*

Here's the full `try` connection code.

```
require_once '../src/DBconnect.php';

$sql = "SELECT *
FROM users
WHERE location = :location";
$location = $_POST['location'];

$statement = $connection->prepare($sql);
$statement->bindParam(':location', $location, PDO::PARAM_STR);
$statement->execute();
$result = $statement->fetchAll();
```

Great, now we have the whole process to retrieve the filtered data. All that's left is to print out the result.

Outside of the `try/catch` connection block and below the header, I'm going to insert the code for a HTML table.

We'll check if this is a POST request, and if the result of our query has more than 0 rows, open the table, loop through all the results, and close the table. If there are no results, display a message. Here's how we will structure the logic:

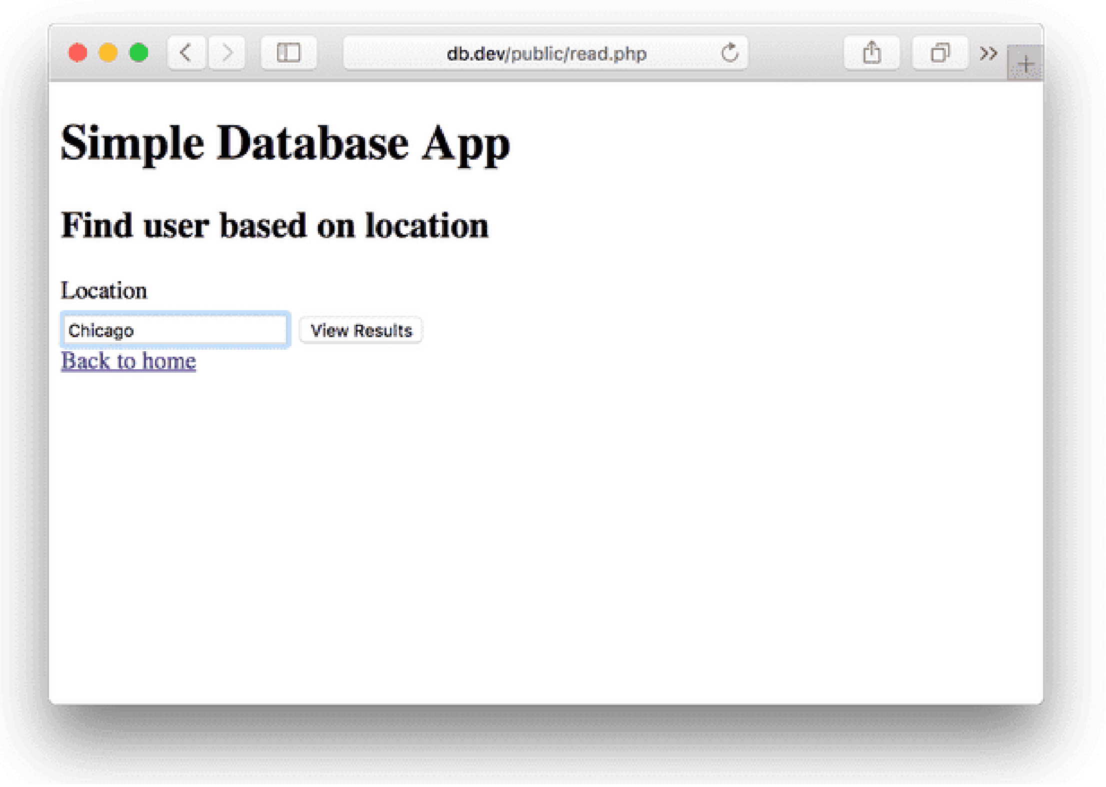
```
if (isset($_POST['submit'])) {
      if ($result && $statement->rowCount() > 0) {
            // open table
            foreach ($result as $row) {
                  // table contents
            }
            // close table
      } else      {
            // no results
      }
    }
```

Here is the final code for **public/read.php**.

```
                                                                 public/read.php
<?php

/**
  * Function to query information based on
  * a parameter: in this case, location.
  *
  */

if (isset($_POST['submit'])) {
  try {
    require "../common.php";
    require_once '../src/DBconnect.php';

    $sql = "SELECT *
    FROM users
    WHERE location = :location";
    $location = $_POST['location'];

    $statement = $connection->prepare($sql);
    $statement->bindParam(':location', $location, PDO::PARAM_STR);
    $statement->execute();
    $result = $statement->fetchAll();

  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }
}

require "templates/header.php";

if (isset($_POST['submit'])) {
  if ($result && $statement->rowCount() > 0) {
?>
    <h2>Results</h2>
    <table>
      <thead>
<tr>
  <th>#</th>
  <th>First Name</th>
  <th>Last Name</th>
  <th>Email Address</th>
  <th>Age</th>
  <th>Location</th>
  <th>Date</th>
</tr>
      </thead>
      <tbody>
  <?php foreach ($result as $row) { ?>
      <tr>
<td><?php echo escape($row["id"]); ?></td>
<td><?php echo escape($row["firstname"]); ?></td>
<td><?php echo escape($row["lastname"]); ?></td>
<td><?php echo escape($row["email"]); ?></td>
<td><?php echo escape($row["age"]); ?></td>
<td><?php echo escape($row["location"]); ?></td>
<td><?php echo escape($row["date"]); ?> </td>
      </tr>
    <?php } ?>
      </tbody>
  </table>
  <?php } else { ?>
    > No results found for <?php echo escape($_POST['location']); ?>.
  <?php }
} ?>

<h2>Find user based on location</h2>

<form method="post">
  <label for="location">Location</label>
  <input type="text" id="location" name="location">
  <input type="submit" name="submit" value="View Results">
</form>

<a href="index.php">Back to home</a>

<?php require "templates/footer.php"; ?>
```

Once there are a few fields in the DB, we can play around with it. Input the location and click ‘view results’.

Search Location:


db.dev/public/read.php

Simple Database App

Find user based on location

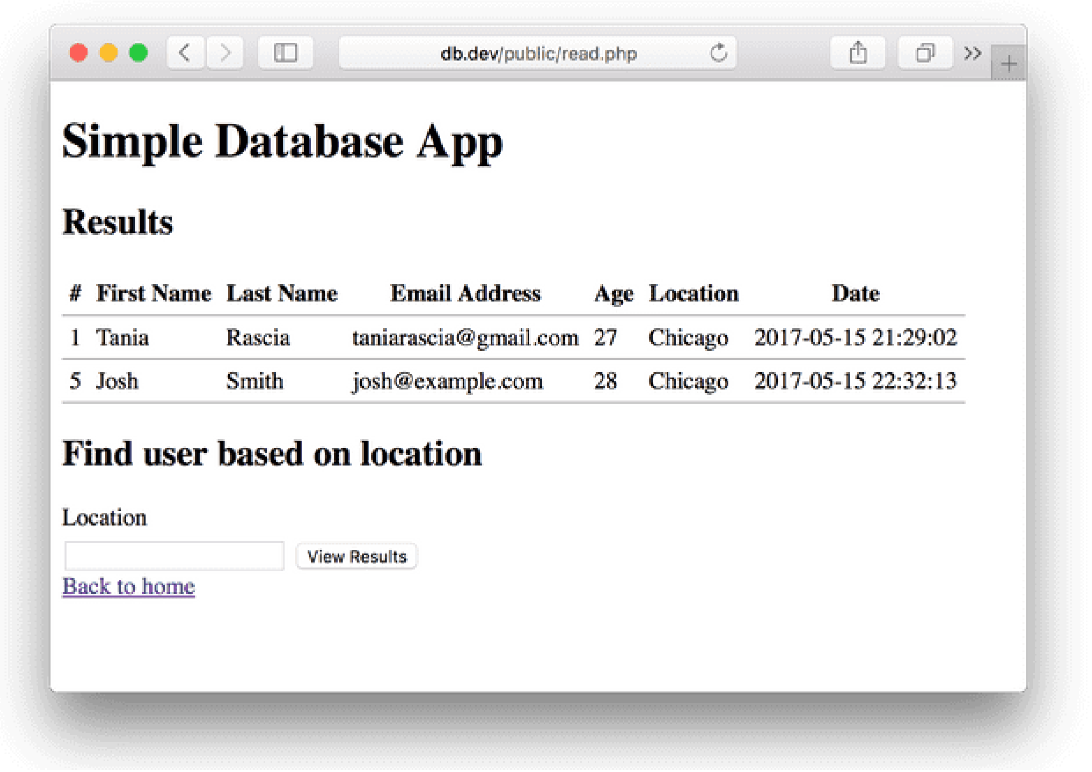
Location

Chicago

View Results

Back to home

And view the results:


db.dev/public/read.php

Simple Database App

Results

#	First Name	Last Name	Email Address	Age	Location	Date
1	Tania	Rascia	taniarascia@gmail.com	27	Chicago	2017-05-15 21:29:02
5	Josh	Smith	josh@example.com	28	Chicago	2017-05-15 22:32:13

Find user based on location

Location

View Results

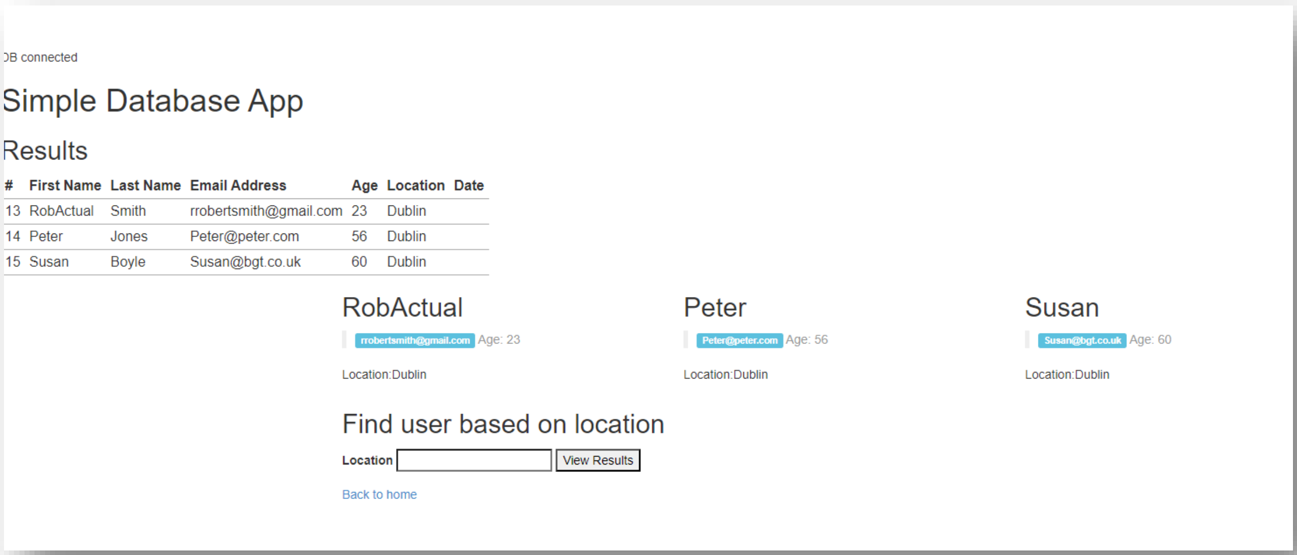
Back to home

Congratulations, you now have the very beginnings of a simple PHP database app without frameworks.

The SymfonyCasts tutorial series uses <div> tags and css to make the output look a lot prettier than ours. You can achieve something similar with the following code (use this instead of the table above. You will also need to use main.css and bootstrap.css from the AirPupnMeow project).

NB: I left the pet references (e.g., $cutePet) on purpose so that you could draw comparisons between this code and the code in the AirPupnMeow project. It would not be OK to do this for any work that you submit to me.

```
<div class="container">
    <div class="row">
        <?php foreach ($result as $cutePet) { ?>
            <div class="col-md-4 pet-list-item">
                <h2>
                    <?php echo $cutePet['firstname']; ?>
                </h2>

                <blockquote class="pet-details">
                    <span class="label label-info"><?php echo
$cutePet['email']; ?></span>
                    <?php echo 'Age: '. $cutePet['age']; ?>
                 </blockquote>
                <p>
                    <?php echo 'Location:'. $cutePet['location']; ?>
                </p>
            </div>
        <?php } ?>
    </div>
```



DB connected

Simple Database App

Results

#	First Name	Last Name	Email Address	Age	Location	Date
13	RobActual	Smith	rrobertsmith@gmail.com	23	Dublin	
14	Peter	Jones	Peter@peter.com	56	Dublin	
15	Susan	Boyle	Susan@bgt.co.uk	60	Dublin	

RobActual

rrobertsmith@gmail.com Age: 23

Location:Dublin

Peter

Peter@peter.com Age: 56

Location:Dublin

Susan

Susan@bgt.co.uk Age: 60

Location:Dublin

Find user based on location

5.1 Directory structure at this point

```
Root_Dir
      |-- config.php
      |-- common.php
      |-- Install.php
      |-- data/
            |-- init.sql
      |-- src/
            |-- DBconnect.php
      |-- public/
            |-- css/
               |-- style.css
            |-- templates/
               |-- header.php
               |-- footer.php
            |-- index.php
            |-- create.php
            |-- read.php
```

This directory structure roughly adheres to then MVC design pattern. If we were using Classes (i.e. better OOPHP), I would expect most of the logic to appear in the src directory.

6 Conclusion

In this tutorial, we went over a lot of valuable lessons, including but not limited to:

- templating (header.php & footer.php)
- connecting to a MySQL database with PDO,
- creating an installer script,
- inserting users into a database (Create),
- selecting and printing out users from a database (Read),
- sanitising user input (server-side).

In part 2 of this tutorial we will focus on the Update and Delete aspects of CRUD.

If this were a real world app, of course there would be more considerations to make. For example the 'backend' would have to be password protected, which means you would make a login page and administrative users who are the only ones who have access to the app.