

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

## -Part 2-

### Table of Contents

1	Preamble.....	1
1.1	PREREQUISITES .....	1
1.2	GOALS.....	2
2	Set up the environment .....	3
3	Create an editable list of all users.....	4
3.1	Using HTTP query strings .....	6
4	Modify an existing user .....	10
5	Deleting entries from a database .....	17
6	Conclusion.....	19

## 1 Preamble

In the first part of this tutorial series, we covered how to connect to a MySQL database with PHP using the modern PDO method. We accomplished this by developing the first half of a very simple CRUD app. CRUD stands for Create, Read, Update, Delete, and it is a common way to store, view, and modify data.

In this part of the tutorial, we will learn how to modify existing entries (update) and permanently remove existing entries (delete).

### 1.1 PREREQUISITES

Before you complete this tutorial, you will need to have completed Part 1: Create, Read. Other prerequisites include:

- **A basic knowledge of HTML** - We'll be using [HTML forms](#).
- **A basic knowledge of PHP** - I'm going to do my best to simplify as much as possible, but you should already have an idea of how variables and strings work. [Here's a basic guide to some of the fundamentals](#).
- **A local PHP and MySQL environment** – Laragon, LAMP, XXAMP etc...
- **A database management app** - You can use HeidiSQL, PHPMysqlAdmin, 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

- Create a page that lists all users with an edit button next to their name
- Dynamically create a unique page for editing the data of any specific user
- Create a page that lists all users with a delete button next to their name

## 2 Set up the environment

If you're following along directly from part one, you will have all the code you need to start ready and functioning.

Everything we will be editing is in the `public/` folder, as all our initial setup is out of the way. Go ahead and check that everything is working from the place we left off in part 1, and then we can continue.

*Side note: Some of the screen shots show the URL <http://db.server>. For the purpose of our tutorial, we will use [localhost](http://localhost) instead.*

### 3 Create an editable list of all users

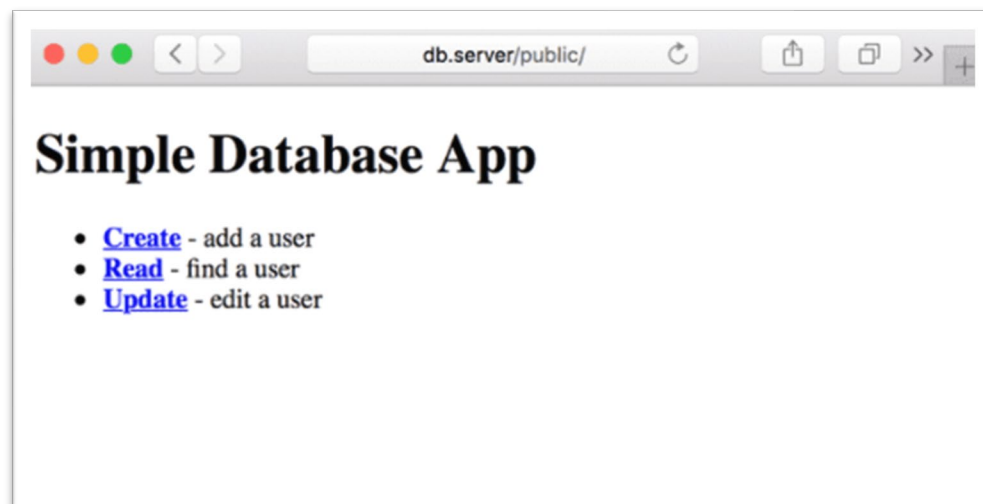
We left off in part one with **create.php** and **read.php**. Now we're going to create two new files. Let's begin by creating **update.php**.

First, in **index.php**, add a link to **update.php**.

```
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>
  <li><a href="update.php"><strong>Update</strong></a> - edit a user</li>
</ul>

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



Now that the main view of our app links to the **update.php**, let's create the file.

The purpose of this file is to list all users in the database, and show an "Edit" link next to each user. The Edit link will allow us to edit each user individually. We will use most of the same code from **read.php**, except more simplified.

We will use a simple **SELECT** statement to get all users.

```
$sql = "SELECT * FROM users";
```

This is the simplest possible SQL command we can execute with PDO - simply select all users, prepare the statement, and store the result in **\$result**.

```
public/update.php
$sql = "SELECT * FROM users";

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

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

Using that, we can build out our try/catch block at the top of **update.php**.

```
public/update.php

<?php

/**
 * List all users with a link to edit
 */

try {
    require "../common.php";
    require_once '../src/DBconnect.php';

    $sql = "SELECT * FROM users";

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

    $result = $statement->fetchAll();
} catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
}
?>
```

Right below this code, we'll print the HTML table with the data from our SELECT statement.

```
public/update.php

<?php require "templates/header.php"; ?>

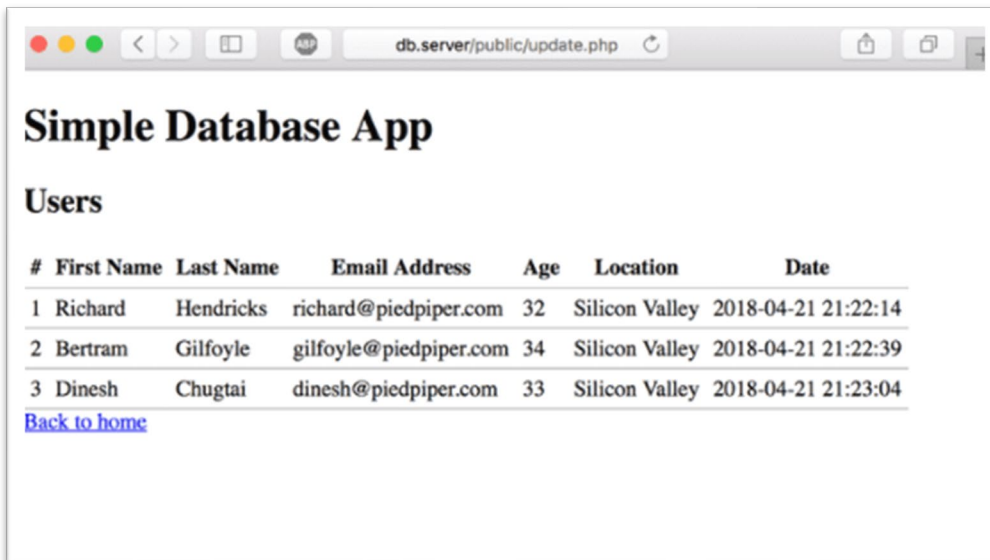
<h2>Update users</h2>

<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 endforeach; ?>
    </tbody>
</table>

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

Before anything will show up here, we'll have to go back to **create.php** and add a few users, which I just did for this example.

Here is what we have now in **update.php**:



#	First Name	Last Name	Email Address	Age	Location	Date
1	Richard	Hendricks	richard@piedpiper.com	32	Silicon Valley	2018-04-21 21:22:14
2	Bertram	Gilfoyle	gilfoyle@piedpiper.com	34	Silicon Valley	2018-04-21 21:22:39
3	Dinesh	Chugtai	dinesh@piedpiper.com	33	Silicon Valley	2018-04-21 21:23:04

*If the date field is empty, do not worry about it for now.*

### 3.1 Using HTTP query strings

Up until now, this is all the same stuff we covered in **read.php**. The interesting part that we'll add now is how we'll get to a page where we can edit each individual user.

Below Date in the table, let's add a column heading (`<th>`) for Edit.

```
<th>Edit</th>
```

Now in the `tbody`, we'll add a link that corresponds to this header for each user. In just a moment, we'll create a new file called **update-single.php**, which will be an edit page for each user. But how will **update-single.php** know if we're editing Richard or Gilfoyle?

We're going to tell **update-single.php** via a parameter in the URL which user to edit. Since we know id of each user is unique, we can safely use id to identify each user.

*Remember that our code queries all rows in the DB and stores the result in `$result`. We then loop through the fields stored in `$result` and display them as `$row[s]` with the help of a `foreach` loop. Try `var_dump[ing] $result` and see what you get. You'll see that every row has an ID.*

*We can grab a specific row by its ID.*

*Remember URL parameters? That's when there is information added to the http server request, e.g., [www.mysite.com/update-single.php?id=3](http://www.mysite.com/update-single.php?id=3). The parameter value '3' allows us to grab the row with the id '3'.*

*`?id=3` is a HTTP query string*

Let's create a link that goes to **update-single.php**, and tack a question mark after the file name, which begins an HTTP query string. After this question mark, we can insert as many key/value pairs as we want in the URL.

*Remember, `$result` is an associative array, so it's full of key/value pairs. Using this information in a HTTP query string is a great way to grab specific DB fields.*

If we want to access Dinesh, the user with an `id` of 3, our url will be `update-single.php?id=3`. We will get those values dynamically the same way we do to just print them out normally, except we'll embed it in the URL, like so:

```
<td><a href="update-single.php?id=?php echo escape($row["id"]);  
?>">Edit</a></td>
```

*See how we have cleverly used the `echo` method to print the row ID (instead of hard coding the ID value). Because this code is in a loop, it will do this for every row... and therefore, every ID.*

Here is the final code for **update.php**.

```
public/update.php

<?php

/**
 * List all users with a link to edit
 */

try {
    require "../common.php";
    require_once '../src/DBconnect.php';

    $sql = "SELECT * FROM users";

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

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

?>
<?php require "templates/header.php"; ?>

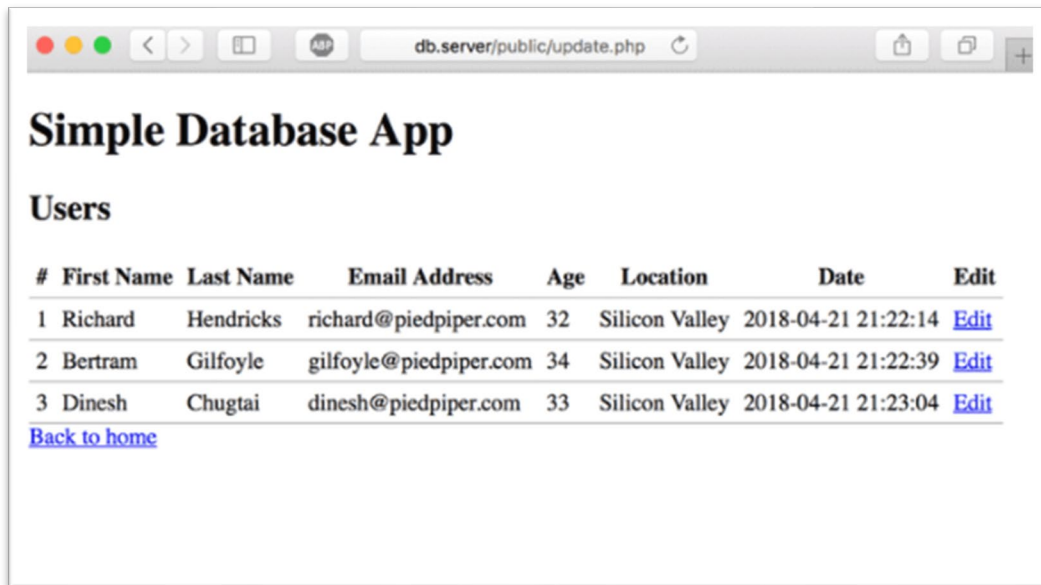
<h2>Update users</h2>

<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>
            <th>Edit</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>
                <td><a href="update-single.php?id=<?php echo escape($row["id"]);
?>">Edit</a></td>
            </tr>
        <?php endforeach; ?>
    </tbody>
</table>

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

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





Now we have a list of all users with links to edit them. If you hover over each edit button, you'll see the id in the URL change to correspond to each user. However, the links don't go anywhere yet, so that's what we'll work on next.

*This type of functionality is useful when working with shopping carts and checkouts for ecommerce sites.*

## 4 Modify an existing user

Editing an existing database entry is the trickiest of all four aspects of CRUD, and it's particularly difficult to find a modern, cohesive tutorial of any sort on how to do this with HTML and PHP, especially using the modern PDO method, so here we go!

*Remember, all we have done so far is to grab a row by its row ID. We've added that ID to a URL... but at the moment, that URL doesn't go anywhere.*

First, we'll want to display the data we're going to edit, then we'll need to add functions to modify it.

In **update.php**, we created edit links that detect the `id` of the users and create a unique URL for each user. In **update-single.php**, we have to figure out which `id` is being loaded in, because all the edit pages will load and be routed through this single page.

Just as we've used the superglobal `$_POST` to detect what data has been posted through an HTML form, we'll use `$_GET` to retrieve information from the URL. It's important to remember that sensitive data such as passwords should never be passed through the `$_GET` variable; however, for our simple purposes today, it will do just fine.

Let's create **update-single.php**, load in the required files, and make an if/else statement. We'll check if `id` is found in the URL, otherwise we'll just show a brief error message and close the script.

```
public/update-single.php
<?php
require "../common.php";

if (isset($_GET['id'])) {
    echo $_GET['id']; // for testing purposes
} else {
    echo "Something went wrong!";
    exit;
}
?>
```

Now if I click on Dinesh...



The page will print out 3, as seen in the URL. Great! Now that we know that's working, we can use it to pull the data specifically for user id 3.

We're still working with `SELECT` statements here, so we're doing exactly what we did previously with the location variable in **read.php**. We'll assign `$_GET['id']` to a variable, bind it to the name of `id` (with `bindValue()`), and look for the `id` with the `WHERE` clause.

*Remember binding the value of the variable `$ID`, to the 'placeholder' `:ID` helps to secure against SQL injection attacks. Why is that?*

```
require "../common.php";
public/update-single.php

if (isset($_GET['id'])) {
    try {
        require_once '../src/DBconnect.php';

        $id = $_GET['id'];

        $sql = "SELECT * FROM users WHERE id = :id";
        $statement = $connection->prepare($sql);
        $statement->bindValue(':id', $id);
        $statement->execute();

        $user = $statement->fetch(PDO::FETCH_ASSOC);
    } catch(PDOException $error) {
        echo $sql . "<br>" . $error->getMessage();
    }
} else {
    echo "Something went wrong!";
    exit;
}
```

Now we want to display the data, but it's going to be a little different than the previous times we printed out data, because we want to be able to update this data as well. How will we `UPDATE` it? With HTML forms and inputs, just like we `INSERT` and `SELECT` data.

Below the database code, we'll begin our view for `update-single.php`.

*Hmm... perhaps the view and the logic should be separated. That would be more in keeping with the MVC design pattern.*

*We leave them both together for this tutorial so that is easier to follow, but you can divide your code by its functionality with the aid of classes and methods.*

Now, since I know the data consists of first name, last name, and so on, I can just manually type it all out as we did before, but there's a more efficient way to get all that data. Let's dynamically print out each data column and value as a key/value pair in PHP.

We'll start by writing a `foreach` loop, but instead of returning the entire associative array in the variable, we'll separate the keys and values into their own variables. We'll get the value from the `fetch(PDO::FETCH_ASSOC)` above. We stored that value in `$user`.

*`PDO::FETCH_ASSOC`: returns an array indexed by column name as returned in your result set*

We're going to put the entire loop inside a form with a submit button (at the top of the **update-single.php** script). Here is the logic we will use:

```
public/update-single.php
<?php require "templates/header.php"; ?>

<h2>Edit a user</h2>

<form method="post">
  <?php foreach ($user as $key => $value) : ?>
    // print data here
  <?php endforeach; ?>
  <input type="submit" name="submit" value="Submit">
</form>

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

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

Inside the foreach, we're making a HTML form which will consist of a `<label>` and `<input>` for each form field. Each label will be a column name from the database.

*Remember, a regular HTML form creates labels and inputs as follows:*

```
<label for="firstname">First Name</label>
```

```
<input type="text" name="firstname" id="firstname">
```

*We are simply creating a template which is populated with each iteration through the loop.*

```
<label for="<?php echo $key; ?>"> <?php echo ucfirst($key); ?> </label>
```

[ucfirst\(\)](#) — Makes a string's first character uppercase

Each value will be the value of an input field (i.e., the input field will not be blank). In the context of key/value pairs in the associative array, we'll be using the key as the name and id of the input, and the value as the value. I'm also adding a ternary (quick conditional statement) to make the input "readOnly" if the key name is id ...because the ID field should not be editable.

```
<input type="text" name="<?php echo $key; ?>" id="<?php echo $key; ?>"
value="<?php echo escape($value); ?>"
<?php echo ($key === 'id' ? 'readonly' : null); ?>>
```

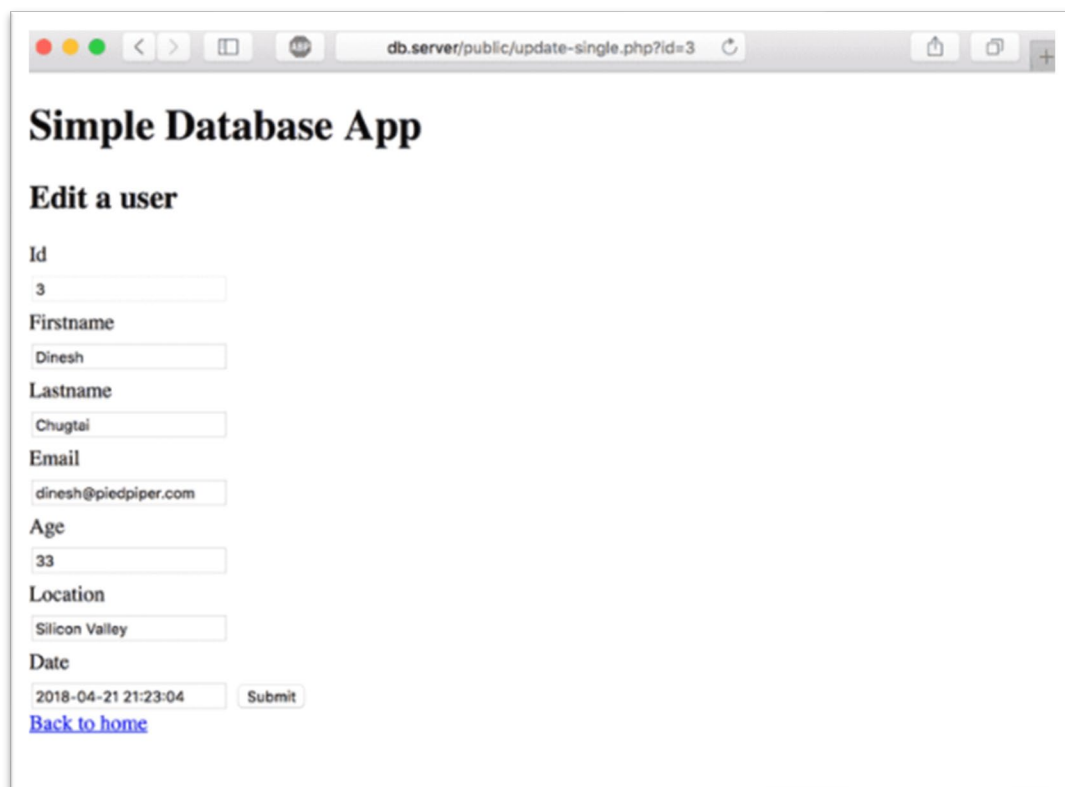
*See how we substitute normal HTML (white text) for dynamic PHP (yellow text), otherwise it is just a standard HTML from `<Input>` field. Actually, it is treated as 'normal' HTML by the browser.*

Now the entire form looks like this.

```
public/update-single.php
<form method="post">
  <?php foreach ($user as $key => $value) : ?>
    <label for="<?php echo $key; ?>"><?php echo ucfirst($key); ?></label>
    <input type="text" name="<?php echo $key; ?>" id="<?php echo $key;
?>" value="<?php echo escape($value); ?>" <?php echo ($key === 'id' ?
'readonly' : null); ?>
    <?php endforeach; ?>
    <input type="submit" name="submit" value="Submit">
  </form>
```

*So you see, we created one dynamic <label> and <input> field and let the foreach loop repeat that for every item in the DB.*

Okay, that was quite a bit of work, but now look what we have! Notice that all of the value fields are already filled in. Any of these can be edited (except ID).



The screenshot shows a web browser window with the address bar displaying 'db.server/public/update-single.php?id=3'. The page title is 'Simple Database App'. Below the title is a section titled 'Edit a user'. The form contains the following fields:

- Id**: A text input field containing the value '3'.
- Firstname**: A text input field containing the value 'Dinesh'.
- Lastname**: A text input field containing the value 'Chugtai'.
- Email**: A text input field containing the value 'dinesh@piedpiper.com'.
- Age**: A text input field containing the value '33'.
- Location**: A text input field containing the value 'Silicon Valley'.
- Date**: A text input field containing the value '2018-04-21 21:23:04'.

At the bottom of the form, there is a 'Submit' button and a blue link labeled 'Back to home'.

The single edit page now has editable fields for each value. However, our code at this point is only to display the data - we need to add another section to process the code after the submit button has been pressed. That is, to write the edited information to the DB.

At the top of **update-single.php**, right below the line that requires common.php, check if the form has been submitted, and begin the try/catch block.

```
if (isset($_POST['submit'])) {  
    try {  
        require_once '../src/DBconnect.php';  
        // run update query  
    } catch(PDOException $error) {  
        echo $sql . "<br>" . $error->getMessage();  
    }  
}
```

We want to use the UPDATE clause to SET each value to the new value. The literal code for our case looks like this:

```
UPDATE users  
    SET id = :id,  
        firstname = :firstname,  
        lastname = :lastname,  
        email = :email,  
        age = :age,  
        location = :location,  
        date = :date  
    WHERE id = :id
```

As before, this SQL query will be stored in a variable. We called the variable `$sql` in previous exercises, so we might as well keep with that trend. We will also need to grab the updated clause from the form through the `$_POST` array, then prepare and execute the SQL query. See the completed code below for the complete solution.

That's all we need to update the values now! I'm just going to quickly add in some code to display that everything has been updated on the front end. I'll add this after the header.php file is required.

```
<?php if (isset($_POST['submit']) && $statement) : ?>  
    <?php echo escape($_POST['firstname']); ?> successfully updated.  
<?php endif; ?>
```

Here is the entirety of the file we just created. It's a long one because we had to get the information (based on ID) from the DB, put that into a form, then allow the user to edit and update the DB again.

```
public/update-single.php
<?php
/**
 * Use an HTML form to edit an entry in the
 * users table.
 *
 */
require "../common.php";
if (isset($_POST['submit'])) {
    try {
        require_once '../src/DBconnect.php';
        $user = [
            "id" => escape($_POST['id']),
            "firstname" => escape($_POST['firstname']),
            "lastname" => escape($_POST['lastname']),
            "email" => escape($_POST['email']),
            "age" => escape($_POST['age']),
            "location" => escape($_POST['location']),
            "date" => escape($_POST['date'])
        ];

        $sql = "UPDATE users
                SET id = :id,
                    firstname = :firstname,
                    lastname = :lastname,
                    email = :email,
                    age = :age,
                    location = :location
                WHERE id = :id";

        $statement = $connection->prepare($sql);
        $statement->execute($user);
    } catch(PDOException $error) {
        echo $sql . "<br>" . $error->getMessage();
    }
}

if (isset($_GET['id'])) {
    try {
        require_once '../src/DBconnect.php';
        $id = $_GET['id'];
        $sql = "SELECT * FROM users WHERE id = :id";
        $statement = $connection->prepare($sql);
        $statement->bindValue(':id', $id);
        $statement->execute();

        $user = $statement->fetch(PDO::FETCH_ASSOC);
    } catch(PDOException $error) {
        echo $sql . "<br>" . $error->getMessage();
    }
} else {
    echo "Something went wrong!";
    exit;
}
?>

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

```

<?php if (isset($_POST['submit']) && $statement) : ?>
    <?php echo escape($_POST['firstname']); ?> successfully updated.
<?php endif; ?>

<h2>Edit a user</h2>

<form method="post">
    <?php foreach ($user as $key => $value) : ?>
        <label for="<?php echo $key; ?>"><?php echo ucfirst($key); ?></label>
        <input type="text" name="<?php echo $key; ?>" id="<?php echo $key;
?>" value="<?php echo escape($value); ?>" <?php echo ($key === 'id' ?
'readonly' : null); ?>>
        <?php endforeach; ?>
        <input type="submit" name="submit" value="Submit">
    </form>

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

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

```

As a test, I updated the age value to make sure it worked.

Simple Database App

Dinesh successfully updated.

### Edit a user

Id  
3

Firstname  
Dinesh

Lastname  
Chugtai

Email  
dinesh@piedpiper.com

Age  
35

Location  
Silicon Valley

Date  
2018-04-21 21:23:04

[Back to home](#)

Now that we've completed the update process, it's time to move on to deleting, which is much simpler.



## 5 Deleting entries from a database

Back in **index.php**, let's add an entry for delete. We could put this in the update file, but for the sake of continuity and finishing the acronym, let's just make it into a new file.

```
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>
  <li>
    <a href="update.php"><strong>Update</strong></a> - edit a user
  </li>
  <li>
    <a href="delete.php"><strong>Delete</strong></a> - delete a user
  </li>
</ul>
```

There's nothing new to learn with the `DELETE` statement. Let's copy the code from **update.php** over, but change the text on the "edit" button to "delete". We'll have the delete button link to the same URL with a query string added, instead of creating a new page for deletion.

```
<td><a href="delete.php?id=<?php echo escape($row["id"]); ?> "> Delete </a>
</td>
```

The `DELETE` statement is similar to a `SELECT` statement, and we'll check for the `$_GET` superglobal again. If the proper id is loaded into the URL, PHP will delete that user with that ID.

```
public/delete.php
<?php
/**
 * Delete a user
 */

require "../common.php";

if (isset($_GET["id"])) {
  try {
    require_once '../src/DBconnect.php';

    $id = $_GET["id"];

    $sql = "DELETE FROM users WHERE id = :id";

    $statement = $connection->prepare($sql);
    $statement->bindValue(':id', $id);
    $statement->execute();

    $success = "User ". $id. " successfully deleted";
```

```

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

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

    $sql = "SELECT * FROM users";

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

    $result = $statement->fetchAll();
} catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
}
?>
<?php require "templates/header.php"; ?>

<h2>Delete users</h2>

<?php if ($success) echo $success; ?>

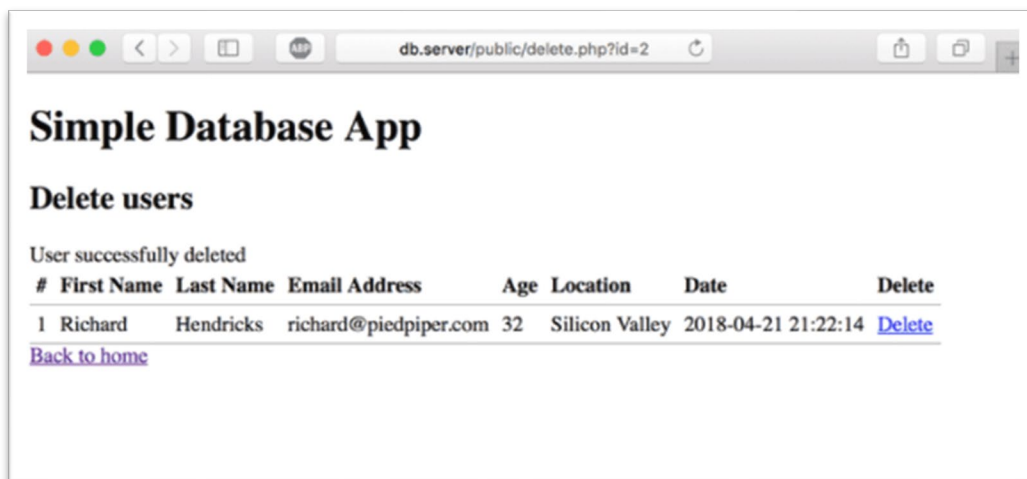
<table>
    <thead>
        <tr>
            <th>#</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email Address</th>
            <th>Age</th>
            <th>Location</th>
            <th>Date</th>
            <th>Delete</th>
        </tr>
    </thead>
    <tbody>
        <?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>
                <td><a href="delete.php?id=<?php echo escape($row["id"]);
?>">Delete</a></td>
            </tr>
        <?php endforeach; ?>
    </tbody>
</table>

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

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

```

Here's the page after I deleted two users.



Often, in a real app, data won't be permanently deleted from the database. The users might have a boolean table that defines them as "active" or "inactive" users instead of actually deleting the data. Nonetheless, DELETE is important to know, and it can be used along with SELECT to delete then insert new data as opposed to updating it with UPDATE.

## 6 Conclusion

That was a lot of information. If you got lost somewhere along the way, I don't blame you! There's a reason few tutorials venture into this territory.

I hope you enjoyed this tutorial - learning how to make your own CRUD application is a lot of work, and difficult. Though being able to access information from outside your application is a fundamental skill for any programmer. Similar processes are used in Java, C# and practically every other programming language.

There is still more to learn for this course, however, at this point you have everything you need to create a fully functional CRUD app. You will need to create several CRUD apps before you become comfortable with the programming logic and workflows. Therefore, I recommend that you repeat this tutorial numerous times with different DB data, e.g., for movies, hardware, or even fashion. Why not try to change the layout to make the site more attractive or try to separate out the code by functionality. Converting the code to a more OOPHP structure would also be useful.

Since I know it needs to be said again, this is not meant to be a complete, secure, production app. However, it's a great introduction and is a minimum viable product that gets PDO, PHP, and MySQL up and running and playing together nicely.