Web Development for Intermediate Programmers - with PHP

Steven Sohcot

Legal

ISBN-13 978-1536819670

Book Version 1.00

Hey, you should read this part!

This book is meant for intermediate programmers. As such, the intended audience will skip the entire "Part 1".

If you plan to do this, I suggest that you at least read the Tip on Page 17 regarding going "in-and-out of PHP".

The source code for this book can be found here:
https://github.com/stevesohcot/php-book

Contents

Part 1: The Basics 8

Chapter 1: Basic Concepts 9
Chapter 2: Sublime Text 25
Chapter 3: HTML Forms 29
Chapter 4: File Inclusions 34
Chapter 5: Database Fundamentals 41
Chapter 6: Connecting to a database 51
Chapter 7: Extracting data from the database 64

Part 2: Making an App 72

Chapter 8: Creating Tables 73
Chapter 9: Adding ("Inserting") Entries into a Database 84
Chapter 10: Viewing the Users 101
Chapter 11: Updating Users - Part 1 110
Chapter 12: Updating Users - Part 2 121
Chapter 13: Deleting Users 128
Chapter 14: Tasks - Part 1 130
Chapter 15: Tasks - Part 2 145
Chapter 16: Code Cleanup - Part 1 155
Chapter 17: Code Cleanup - Part 2 165
Chapter 18: User Validation - Client Side (JavaScript) 168
Chapter 19: User Validation - Server-Side (PHP) 180
Chapter 20: Task Validation - Client Side (JavaScript) 183
Chapter 21: Task Validation - Server-Side (PHP) 187
Chapter 22: User Sign Up 193
Chapter 23: User Login 207

Chapter 24: Remember Me 215
Chapter 25: User Security - Part 1 226
Chapter 26: User Security - Part 2 232

Part 3: Ajax 235

Chapter 27: Ajax - Introduction and Adding Tasks 236
Chapter 28: Viewing Tasks with Ajax 254
Chapter 29: Updating Individual Fields with Ajax 260
Chapter 30: Updating Multiple Fields with Ajax 276
Chapter 31: Deleting with Ajax 288

Part 4: Other Useful PHP Tips 292

Chapter 32: Security Precaution 293
Chapter 33: Miscellaneous PHP Tips 296

Who is this book for?

This book will teach you how to make database-driven web pages using PHP.

Basic web pages are **static**: they do not change. The author will post the content online, and that's it. Dynamic web pages have content that does change – depending on how the user got *to* the page.

This book is not intended for completely inexperienced programmers. A former professor of mine once said "all programming is the same, just a different syntax." Maybe you know how to make web pages in ASP or Ruby on Rails; but you want to learn PHP. Perhaps you have the ability to pick up new concepts quickly. <u>I'm going to assume you know what a "variable" is, and what an "if/then" statement is.</u>

There are many "advanced" methods of coding that require you to install additional software on your computer. I don't use them, so I will not be going over them.

For the readers who have database experience: I'm using a relational database taking a procedural-style approach. If you do not know what the previous sentence means, don't worry!

There is a concept of "design patterns" which means that people share their best practices. I'm doing that here.

How is this book different than other books?

Some of the PHP books I have read:

- Do not include any real-world examples, or they don't include any "useful" code – only theory (how to do something, but not *why* you are doing it)
    - Instead, we're going to build an application step-by-step
- Require you to install some "nice to have" software aside from the bare minimum. The installation is sometimes confusing and could deter users from even proceeding.
    - From my point of view, you need a text editor, and a way to upload files to the Internet. That's it!
- Includes confusing code
    - PHP comes in two forms: one is rather advanced and can intimidate beginners (it's called "object-oriented"). The *other* form is the one covered in this book.

Why read *this* book?

I spent a little more than 10 years with PHP. Then I switched to Ruby on Rails for two years, where I learned a lot of concepts that I have applied to my PHP skills. Having a hybrid of knowledge, I've developed best practices - *for me*.

There are hundreds of ways to code the same thing. Some ways are better than others. I'm going to share what works for me. Maybe you have a better way (style of coding). My goal is to spark ideas in how to improve your code; even if that means copying mine.

In addition, I feel that a lot of PHP books focus *only* on PHP (as expected). Since the publication of these books, another website-building skill has risen: "usability" (i.e. how people interact with the website – the "little" things that might annoy them if not done correctly). I have sprinkled in a variety of usability tips while building our application (*some* are explicitly labelled as such).

Part 2: Making an App

The source code for this book can be found here:
https://github.com/stevesohcot/php-book

Chapter 8: Creating Tables

We are going to build a "simple" application: **A To-Do list with multiple users**.

We need two database tables:

Table called "Users"

Field Name	Data Type	Notes
id	Integer (auto-increments)	Primary Key (unique identifier)
first_name	Text (varchar – 25)	
last_name	Text (varchar – 25)	
email	Text (varchar – 100)	
password	Text (char – 32)	Yes, 32 characters; explained after

Table called "Tasks"

Field Name	Data Type	Notes
id	Integer (auto-increments)	Primary Key (unique identifier)
user_id	Integer	Which user is the task for?
description	Text (varchar – 25)	
date_assigned	Date/Time	
completed	Text (char – 1)	Values will be "Y" or "N"

Note that we have two database tables: it is not efficient to have one *giant* table with *all* the data. There would be too much repetitive information. Instead, we will have two tables and "link" them together. MySQL (the database we are using) is a "relational" database – these are tables that have a relation.

The fields of the tables should be pretty self explanatory, but a couple of notes:

- Both tables have a field called "ID", which is an auto-incrementing integer (whole number)
- The "Tasks" table has a field "User ID" which indicates the user (from the Users table) that the task is for
  - Ideally we would tell the database that there is an "official" link between the two tables (called a "Foreign Key"), but we are going to skip this for now
- Task Completed: this does not have to be a "text" data type. There may be other appropriate data types (such as "boolean", which was not reviewed), but we are sticking with this

You could make the tables "manually" using phpMyAdmin , but I personally like to get a "fresh start" on my databases when developing them. Quite often during development, I'll clear out the entire contents of the database. Therefore I like to write a script (that I can easily run, often) to regenerate the tables. Whenever you need to reset the database, you just need to visit the specific web page.

The syntax for creating a database table is:

```
CREATE TABLE table_name  (
[field one name] [field one data type][other field one attributes],
[field two name] [field two data type][other field two attributes]
)
```

Details on the "Data Types" can be found in Chapter 5, "Database Fundamentals".

This is how we will create the two tables:

create_tables_01.php

```
<?php include_once("a_first.php");?>

<?php

$query = "
  CREATE TABLE Users (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `first_name` VARCHAR(25) NOT NULL ,
  `last_name` VARCHAR(25) NOT NULL,
  `email` VARCHAR(100) NULL ,
  `password` CHAR(32) NULL
  ) ENGINE = InnoDB;
";

if (mysqli_query($db1, $query)) {
  print ("Table: Users - success<br />");
} else {
  print ("<strong>ERROR: Table: Users -   "
                  . mysqli_error($db1) . "</strong><br />");
}


$query = "
  CREATE TABLE Tasks (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `user_id`INT NOT NULL ,
  `description` VARCHAR(100) NOT NULL ,
  `date_assigned` DATETIME NOT NULL ,
  `completed` CHAR ( 1 ) NOT NULL DEFAULT 'N'
  ) ENGINE = InnoDB;
";

if (mysqli_query($db1, $query)) {
  print ("Table: Tasks - success<br />");
} else {
  print ("<strong>ERROR: Table: Tasks -   "
                  . mysqli_error($db1) . "</strong><br />");
}

?>

<?php include_once("a_last.php");?>
```

Figure 8.1 – Creating database tables

First we have a block of code that creates the "Users" table, and then a separate block of code to create the "Tasks" table.

Some notes on the fields for the Users table:

- "id" is an Integer (whole number). It cannot be null. "Null" is another way of saying "nothing", so in this case we are saying that there has to be a value. It will auto-increment automatically whenever there is a new entry in the database. Finally, it is set as the Primary Key, meaning that it is the unique identifier for each row in this table.
- The "first_name", "last_name", and "email" are text fields. The data type is "varchar" meaning that if the value has extra spaces at the end, those spaces will automatically be removed. The length of the fields have a limit (25 or 100 characters). Both are "not null" (meaning, they *are* required).
- The "password" field is also a text field, but it is of data type "char", because it will (*eventually*) always have exactly 32 characters (more on this later).
- The "email" and "password" fields do not have a "not null". Instead it is "null" – meaning that it allows for a blank. We will be slowly adding information into the database, and in the interest of teaching we will skip-over these fields at first. *When you eventually make a "real" application, these fields will most likely also be "not null".*

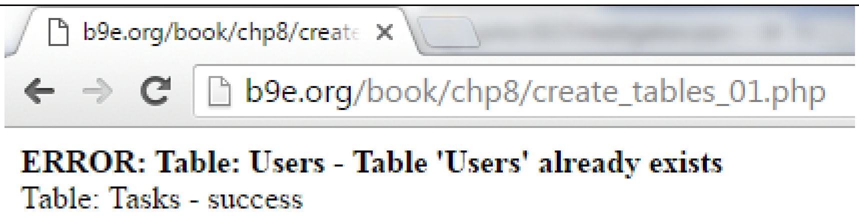
Some notes on the fields for the Tasks table:

- "id" – same exact notes as the "Users" table.
- "user_id" – this will hold the ID number for each User. Yes, just the *number.*
- "completed" – this will be *one* character, and the default (should we forget to put in a value) will be "N" (as in, "No, the brand new task we entered is not yet complete").
    - We could have made it a "varchar(3)" and held the values "Yes" and "No" in the table, but by making it a "char(1)" instead, it will take up less room (megabytes), in turn making the database slightly faster in the long- run.

We have two instances of **`mysqli_query()`** that actually executes the SQL statements. It is surrounded in an IF statement to tell us *which table* was successful in being created, or failed. If the SQL ran correctly, we are given a "success" notification. If it failed, the text "ERROR" will be printed out, in addition to *why* it failed at executing.

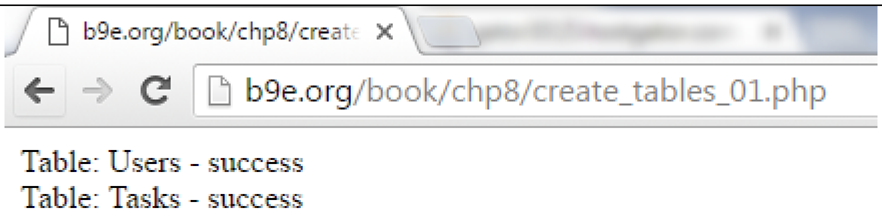
Once you upload this script to your web server, and access it by going to the correct URL, the two tables will be created (or, you will see why there was an error).

Note: From the previous example in Chapter 7, we already have a table called "Users". You have to delete (or in SQL terms, "drop") that table first. You can do this from the phpMyAdmin. If you do not drop (delete) this table first, then you will get an error that the table already exists. If you get this error, it's okay! In the next section, we are going to further automate what we did.



```
ERROR: Table: Users - Table 'Users' already exists
Table: Tasks - success
```

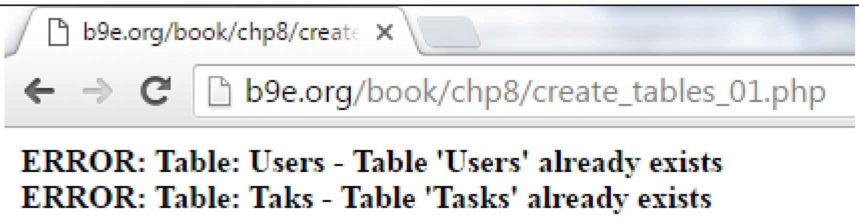
Figure 8.2 – If you do not delete ("drop") the "Users" table first, you will get an error



```
Table: Users - success
Table: Tasks - success
```

Figure 8.3 – If you drop the "Users" table before running this script, you will not get an error

As previously mentioned, I personally like to (quite often) clear out the contents of my databases. If you were to re-run the previous script, you will get an error that both tables exist:



```
ERROR: Table: Users - Table 'Users' already exists
ERROR: Table: Taks - Table 'Tasks' already exists
```

Figure 8.4 – You can't create a table if it already exists

To "clear out" or "reset" the database, we have to:

1. Delete the existing tables (using a SQL "DROP" statement)
2. Re-run the same script that created the tables in the first place

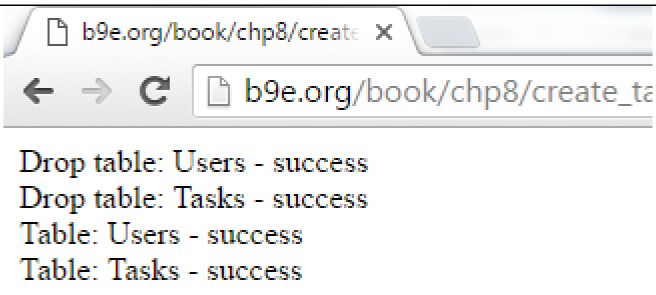
Add in the lines in bold below:

create_tables_02.php

```
<?php include_once("a_first.php");?>

<?php

// First delete the tables
$query = "DROP TABLE Users ";
if (mysqli_query($db1, $query)) {
      print ("Drop table: Users - success<br />");
} else {
      print ("<strong>ERROR: Drop table: Users -  "
                        . mysqli_error($db1) . "</strong><br />");
}


$query = "DROP TABLE Tasks ";
if (mysqli_query($db1, $query)) {
      print ("Drop table: Tasks - success<br />");
} else {
      print ("<strong>ERROR: Drop table: Tasks -  "
                        . mysqli_error($db1) . "</strong><br />");
}

// Then re-create the tables

$query = "
  CREATE TABLE Users (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,

...
```

Figure 8.5 – Drop the tables first

We are running two "DROP" statements (which deletes the tables). After you upload the file, go back to the URL (hitting "refresh" on the page you were on), and you will see that first the two tables are removed *then* the two tables are recreated.



```
Drop table: Users - success
Drop table: Tasks - success
Table: Users - success
Table: Tasks - success
```

Figure 8.6 – Output of Dropping and then Creating the tables (from Figure 8.5)

The author's personal best practice

We are starting to have a lot of repetitive code!

I made a function called "ResetDatabase", that runs a SQL statement (you pass in the database connection and SQL query - as you would have to do anyways). But then I *also* pass in two other parameters: *$query_type* (to indicate if I'm creating or deleting a table), as well as *$table_name*.

```
function ResetDatabase($db_connection, $query_type, $table_name, $SQL) {

  if (mysqli_query($db_connection, $SQL)) {
      print ("$query_type query for $table_name - success<br />");
  } else {
      print ("<strong>ERROR: $query_type for $table_name failed: "
                        . mysqli_error($db_connection)
                        . "</strong><br />");
  }

}
```

Figure 8.7 – PHP Function to create/drop tables

For deleting tables:

I put the name of the table in a variable. Then I can copy/paste the same two lines to generate the SQL statement, and to execute it. Again, when I execute it, I'm passing in information to run the SQL *and* to print out the status (success/failure) for *which table*.

For creating tables:

It is similar to deleting tables. Start by putting the table name in a variable, and update the SQL to reflect the table name:

Instead of: "`CREATE TABLE Users (…`"

It becomes: "`CREATE TABLE $table_name (…`"

Then copy/paste the `ResetDatabase()` function over-and-over.

When you run it in your browser, you should see the same results (successfully dropped two tables and then created two tables). For this specific example with two tables, it may not seem like you are saving on the number of lines of code, but for larger databases (with *many more tables*) this will certainly help. Figure 8.8 has the updated code:

create_tables_03.php

```
<?php include_once("a_first.php");?>

<?php

 function ResetDatabase($db_connection, $query_type, $table_name, $SQL) {

   if (mysqli_query($db_connection, $SQL)) {
      print ("$query_type query for $table_name - success<br />");
   } else {
      print ("<strong>ERROR: $query_type for $table_name failed: "
                          . mysqli_error($db_connection)
                          . "</strong><br />");
   }

 }

// ************************
// Drop existing tables
// ************************

$table_name = "Users";
$query = "DROP TABLE $table_name ";
ResetDatabase($db1, "Drop", $table_name, $query);

$table_name = "Tasks";
$query = "DROP TABLE $table_name ";
ResetDatabase($db1, "Drop", $table_name, $query);
```

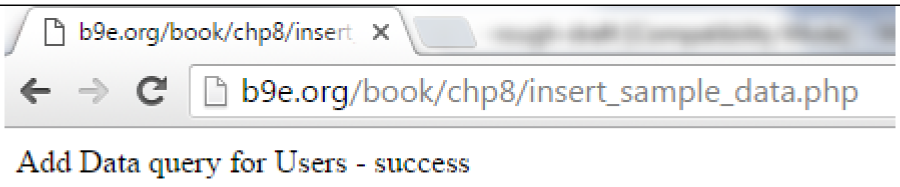
```
// ***********************
// Create tables
// ***********************

$table_name = "Users";
$query = "
      CREATE TABLE $table_name (
            `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
            `first_name` VARCHAR(25) NOT NULL ,
            `last_name` VARCHAR(25) NOT NULL ,
            `email` VARCHAR(100) NULL ,
            `password` CHAR(32) NULL
      ) ENGINE = InnoDB;
";

ResetDatabase($db1, "Create", $table_name, $query);

$table_name = "Tasks";
$query = "
      CREATE TABLE $table_name (
            `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
            `user_id` INT NOT NULL ,
            `description` VARCHAR(100) NOT NULL ,
            `date_assigned` DATETIME NOT NULL ,
            `completed` CHAR ( 1 ) NOT NULL DEFAULT 'N'
      ) ENGINE = InnoDB;
";

ResetDatabase($db1, "Create", $table_name, $query);


?>

<?php include_once("a_last.php");?>
```

Figure 8.8 – Updated code for resetting a database

Note: In future projects, you may want to pre-populate the database with some sample data. You can still use this function! Just change the SQL to an "`INSERT`" statement, and indicate that you are "adding data" for the *$query_type* . Note the updates in bold in Figure 8.9:

```
<?php include_once("a_first.php");?>

<?php

 function ResetDatabase($db_connection, $query_type, $table_name, $SQL) {

       // function goes here, removed for publication purposes

 }
// ***********************
// Drop existing tables
// ***********************

      // code here

// ***********************
// Create tables
// ***********************

      // code here

// ***********************
// Insert Sample Data
// ***********************

$table_name = "Users";
$query = "
      INSERT INTO $table_name
                  (`id`, `first_name`, `last_name`)
      VALUES      ('', 'Peter','Griffin'),
                  ('','Phil','Fry')

";
ResetDatabase($db1, "Add Data", $table_name, $query);

?>

<?php include_once("a_last.php");?>
```

Figure 8.9 – Inserting sample data



```
Add Data query for Users - success
```

Figure 8.10 – Output for inserting sample data (from Figure 8.9)

Note:
Not discussed are Foreign Keys, where you can "officially" tell the database the relationship between tables. If you have Foreign Keys, you may have to change the order in which you create or drop tables: if there is one table that is relying on another table, you will not be able to create/delete it until you take care of the "source" of it. Similarly, when you are pre-populating the database with sample data, you may need to insert data into the specific tables in a certain order.

In the source code provided, I have separated out the files as create_tables_01.php, create_tables_02.php, and create_tables_03.php. You obviously only need one file; perhaps call it just "create_tables.php" and remove the other files.

Warning
Be sure to delete this file from the server after you run it (at least in production)! You don't want to accidentally access this file (or worse, have someone else know of its existence) – otherwise you/anyone could clear-out the contents of the database! I tend to always delete this file after using it; and just re-upload it to the web server as-needed.

Chapter 9: Adding ("Inserting") Entries into a Database

Using knowledge (and copying code!) from Chapters 4 and 7, create four files that will serve as a basic "shell" for the website:

- a_first.php – has connection to the database and includes the "layouts.php" file
- a_last.php – closes the connection to the database
- layouts.php – has two functions:
  - Template_Header() – opening HTML and BODY tags, reference to Bootstrap framework
  - Template_Footer() – closing HTML and BODY tags
- users.php – includes the necessary files, with room for where unique content goes

a_first.php

```
<?php
      ob_start();

      // *****************************
      // START Database connection
      // *****************************
      $host = "localhost";
      $user = "user-name-here";
      $password = "password-here";

      // connect to the database server
      $db1 = mysqli_connect($host, $user, $password)
                  or die("Could not connect to database");

      // select the right database
      mysqli_select_db($db1, "steveb9e_MyDatabaseName");

      // *****************************
      // END Database connection
      // *****************************

      include "layouts.php";
?>
```

Figure 9.1 – PHP code at the start of every page

a_last.php

```
<?php
      // close the connection to the database
      mysqli_close($db1);

      ob_end_flush();
?>
```

Figure 9.2 – PHP code at the end of every page

These two lines may not be familiar:

- The *very first line* of a_first.php: **`ob_start();`**
- The very *last line* of a_last.php: **`ob_end_flush();`**

We will need them later in the book; for now, just put them in. It will help to ensure the page is able to correctly display the results from the database.

layouts.php

```
<?php function Template_Header() { ?>

<!DOCTYPE html>
<html>
<head>
      <title><?php print constant('PAGE_TITLE');?></title>

      <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/
         3.3.7/css/bootstrap.min.css" />

</head>
<body>

      <div class="container" style="margin-top:20px;">
            <ul class="nav nav-pills">
              <li role="presentation" class="active">
                  <a href="users.php">Users</a>
              </li>
              <li role="presentation">
                  <a href="tasks.php">Tasks</a>
              </li>
            </ul>
      </div>

<?php } ?>

<?php function Template_Footer() { ?>

      </body>
      </html>
<?php } ?>
```

Figure 9.3 – PHP functions for the layout of every page

users.php

```
<?php include_once("a_first.php");?>

<?php define("PAGE_TITLE", "Users");?>

<?php Template_Header();?>

      <div class="container">

            <h1>Users</h1>

      </div>

<?php Template_Footer();?>

<?php include_once("a_last.php");?>
```

Figure 9.4 – Display information about the Users in the database

When you open users.php, it will look like:

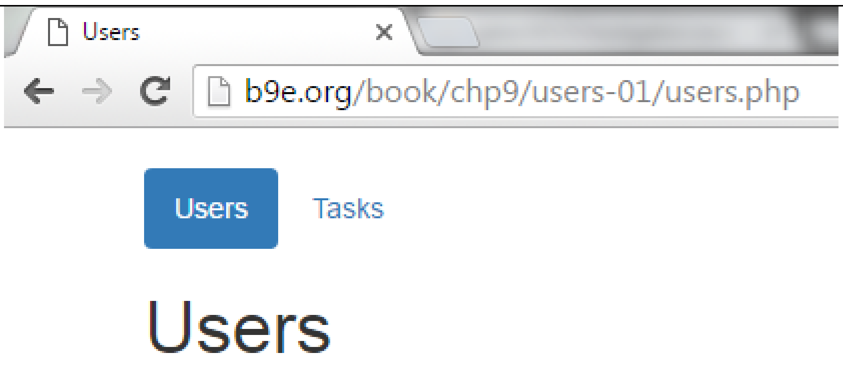


Figure 9.5 – "Users" page, based on Figures 9.1 – 9.4

Let's make a form to add users to the database. Update users.php to:

users.php

```
<?php include_once("a_first.php");?>

<?php define("PAGE_TITLE", "Users");?>

<?php Template_Header();?>

<div class="container">

 <h1>Users</h1>
 <div class="well">
 <h3>Add User</h3>

 <form action="controller_users.php" method="post" class="form-inline">

  <div>First Name:</div>
  <div><input type="text" name="first_name" class="form-control" /></div>

  <div>Last Name:</div>
  <div><input type="text" name="last_name" class="form-control" /></div>

  <div style="margin-top:10px;">
    <input type="hidden" name="AddUser" value="1" />
    <input type="submit" name="btnAddUser"
            value="Add User" class="btn btn-primary" />
  </div>

 </form>

</div> <!-- /.well -->

</div> <!-- /.container -->

<?php Template_Footer();?>

<?php include_once("a_last.php");?>
```

Figure 9.6 – Updated users.php

Note: to start with, we will only populate the First Name and Last Name fields. *For now,* we will skip the "Email" and "Password" fields. This is why we made these (2) fields *allow* for a null (empty) value in the previous chapter.

It should look like:

users.php

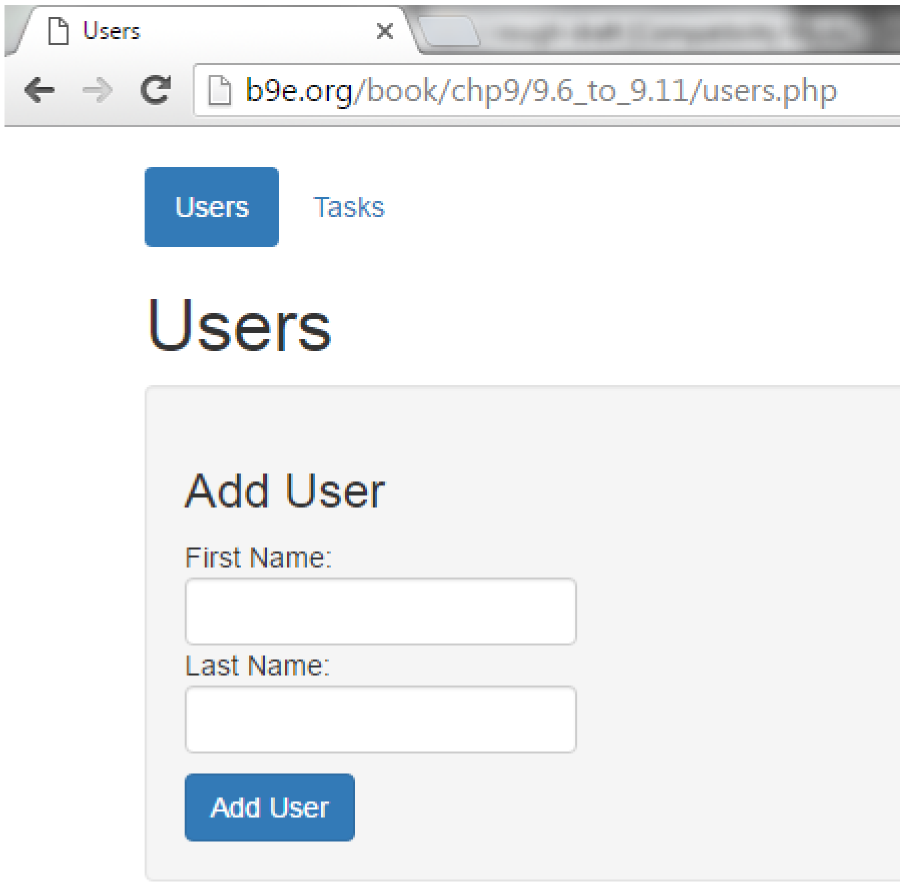


Figure 9.7 – Updated users.php (from Figure 9.6)

A summary of the form components:

- The ACTION of the form goes to "controller_users.php" (we'll make this next)
- There are two text fields (with a *name* of "first_name" and "last_name")
- There is a hidden field with a *name* of "AddUser"
- There is a submit button with the *name* of "btnAddUser"

Also note that some of the HTML tags have Bootstrap classes ("container", "well", "form-inline", "form-control", "btn", "btn-primary"), just to make the examples nicer to look at.

The author's personal best practice

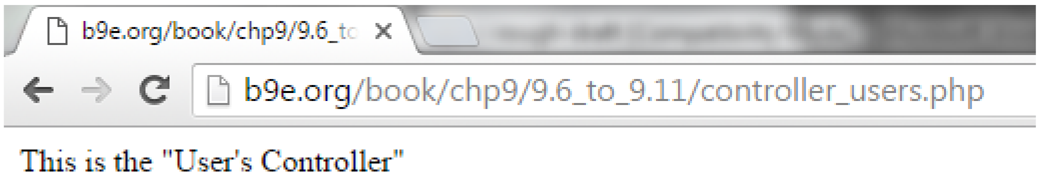
Once the user fills out the form and presses the submit button ("Add User"), the user is sent "to" the page listed in the ACTION of the form (here, "controller_users.php"). Let's create that page now. Make a new file called "controller_users.php"- and the only thing we need in here to start with are the a_first.php and a_last.php files. We are calling this a "controller" because of the MVC approach mentioned in Chapter 1.

controller_users.php

```
<?php include_once("a_first.php");?>

      This is the "User's Controller"

<?php include_once("a_last.php");?>
```

Figure 9.8 – User Controller

Visit this page in your browser, and you will see the text there:



This is the "User's Controller"

Figure 9.9 – Viewing the User's "Controller" (from Figure 9.8)

Let's add some code to controller_users.php:

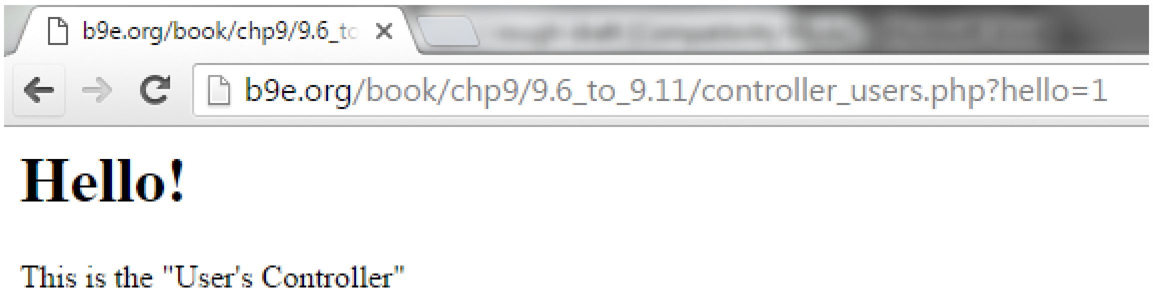
controller_users.php

```
<?php include_once("a_first.php");?>

      <?php

            if (array_key_exists('hello', $_GET)) {
                  print "<h1>Hello!</h1>";
            }

      ?>

      This is the "User's Controller"

<?php include_once("a_last.php");?>
```

Figure 9.10 – User Controller (updated)

The code in bold says "if there was a GET parameter (which comes from the URL, as opposed to a POST parameter) that is called "hello", then print out an H1 tag that says "Hello!".

Refresh the page. You won't see any changes! Now, change the web address to pass in the "hello" parameter in the URL. After the ".php" (in "controller_users.php"), add in **?hello=1** . The value could really be anything (here, it is just "1").



b9e.org/book/chp9/9.6_to_9.11/controller_users.php?hello=1

Hello!

This is the "User's Controller"

Figure 9.11 – User Controller (updated) – with a URL parameter called "hello"

The page detected that there was a variable from the URL called "hello", and it executed the additional PHP code.

This "controller_users.php" page is going to have all of the code for adding, updating, and deleting users – all in the same file. We only want to run the appropriate code: for example, if you are adding a user, then you only want to run the piece of code that adds a user, and no other section of code.

We are going to change our example from "hello" to "AddUser" – which was the name hidden field we had in our form. We are also going to change the **$_GET** to **$_POST** (to match the form's method, POST).

Now, when the user goes to the form on users.php and presses the "Add User" button, they will be brought to controller_users.php and we can run a specific snippet of code: the specific code that will add a user to the database.

Now we just have to:

- Find out the values for "first_name" and "last_name" that the user entered in
- Write a SQL statement to insert the new row (user) into the database table
- Let the user know the action was successfully performed

Here is the code to do this:

controller_users.php

```
<?php include_once("a_first.php");?>

<?php

if (array_key_exists('AddUser', $_POST)) {

      $first_name = $_POST['first_name'];
      $last_name  = $_POST['last_name'];

      $query = "INSERT INTO Users
                  (`id`, `first_name` ,`last_name`)
                  VALUES ('', '$first_name' , '$last_name')";

      if (mysqli_query($db1, $query)) {
            header("Location: users.php?action=user-added");
            exit();
      } else {
            print ("<strong>ERROR: "
                        . mysqli_error($db1) . "</strong><br />");
      }

}

?>

      This is the "User's Controller"

<?php include_once("a_last.php");?>
```

Figure 9.12 – User Controller (updated)

I like to first get all of the user inputs and put them in separate variables:

```
      $first_name = $_POST['first_name'];
      $last_name  = $_POST['last_name'];
```

Then we have the SQL statement that inserts rows into the database (using the syntax found in Chapter 5, "Database Fundamentals"). The final output will substitute the *actual* values from the variables `$first_name` and `$last_name`:

```
$query = "INSERT INTO Users
            (`id`, `first_name`,`last_name`)
            VALUES ('', '$first_name', '$last_name')";
```

Note the tick marks that surround the field names (on a keyboard, this is under the Esc key at the top left, next to the "1" key). This is not required, but depending on the name of the table field you may get an error if it is not there. Also note, the "ID" is blank: the database table is set to populate this field automatically by auto-incrementing the value by "1" from the latest entry.

If the user typed the first name of "Steve" and the last name of "Sohcot" in the form, the SQL query that will *actually* run will be:

```
INSERT INTO Users
   (`user_id`, `first_name`,`last_name`) VALUES ('', 'Steve', 'Sohcot')
```

Further explanation of the code:

`mysqli_query($db1, $query)` is what actually *runs* the SQL statement . It is inside of an IF statement to get confirmation that it works (or we will see why there was an error).

`header("Location: users.php?action=user-added");` will redirect the user to a new page, and **`exit();`** will stop the script from running. We are taking the user back to the "users.php" page <u>but we are adding a variable in the URL</u>, ("action") and we are giving it a specific value ("user-added"). The final result is being brought *back* to the users.php (as expected), and the URL having "action=user-added" appended at the end of it.

Because of the redirect/exit functions we did <u>not</u> see the text "this is the user's controller" as we previously did.

The author's personal best practice

Note that it is possible to have all of the code that adds a user (and displays them) in users.php. However, if the user hits the "refresh" button on the browser after submitting a form, they will get an alert with something along the lines of "Are you sure you want to resubmit the form?". If they hit "yes," then the form is re-submitted and you will be adding a user into the database again. By using this technique of having two separate files (that redirects the back to the initial page), people can hit "Refresh" and the form will not be resubmitted.

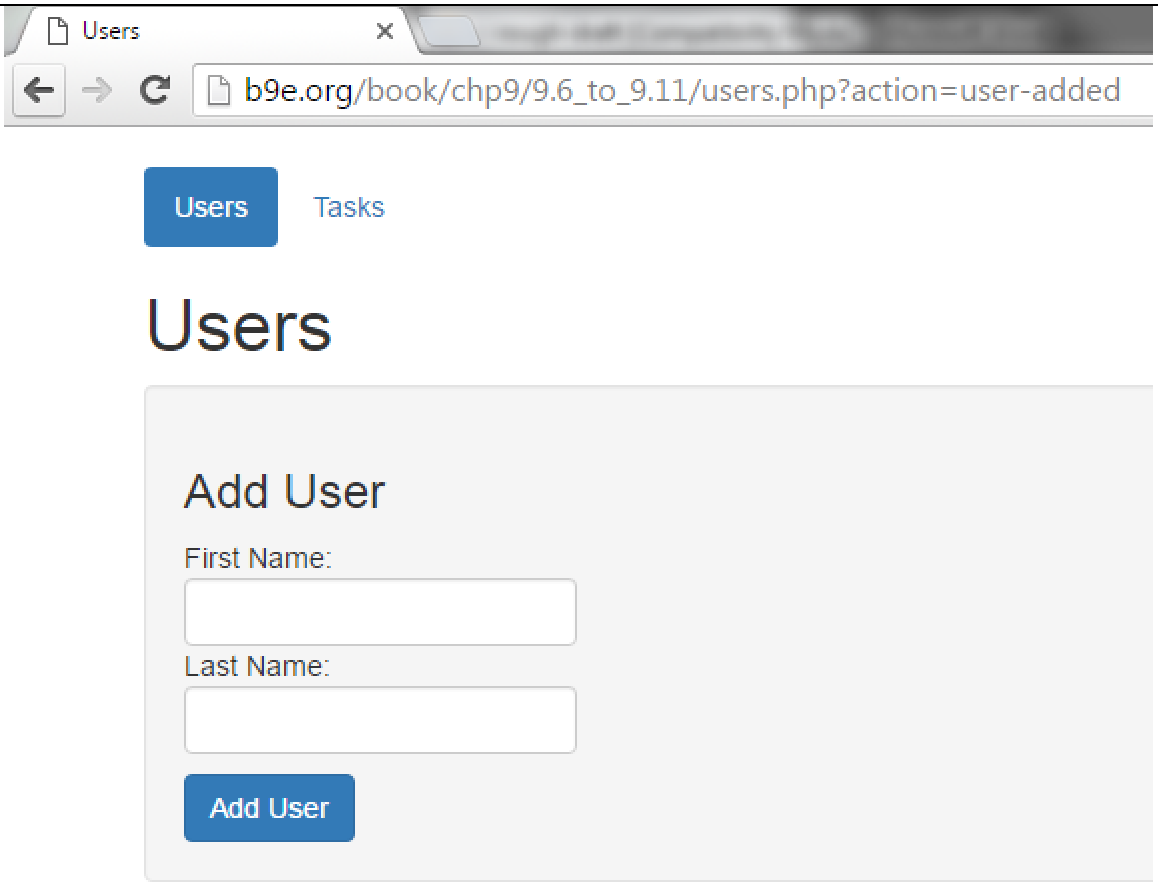


Figure 9.13 – User is brought back to "users.php" with additional variables in URL

We are going to use the URL parameter to show the user a notification indicating the requested action was successful.

We need to make a new file called "alerts.php". We are going to do something similar to controller_users.php where we're running a certain piece of code depending on the variables passed to the page. This page (alerts.php) will eventually have all possible notifications we want to display throughout the application.

alerts.php

```
<?php if ($_GET['action'] == "user-added") { ?>
      <div class="alert alert-success" style="margin-top:10px;">
            The user was successfully added!
      </div>
<?php } ?>

<?php if ($_GET['action'] == "blah") { ?>
      <div class="alert alert-danger">
            You won't see this code in the "blah" section
      </div>
<?php } ?>
```

Figure 9.14 – Displaying possible notifications

The first line detects if there was a GET variable (as in, from the URL) called "action" and if it has the value of "user-added" – which is what we do have. As such, it is going to add the DIV found in that code block. We will eventually have other notifications for the user; but unless the parameter and values given in the URL match what is specified, that piece of code will not be run.

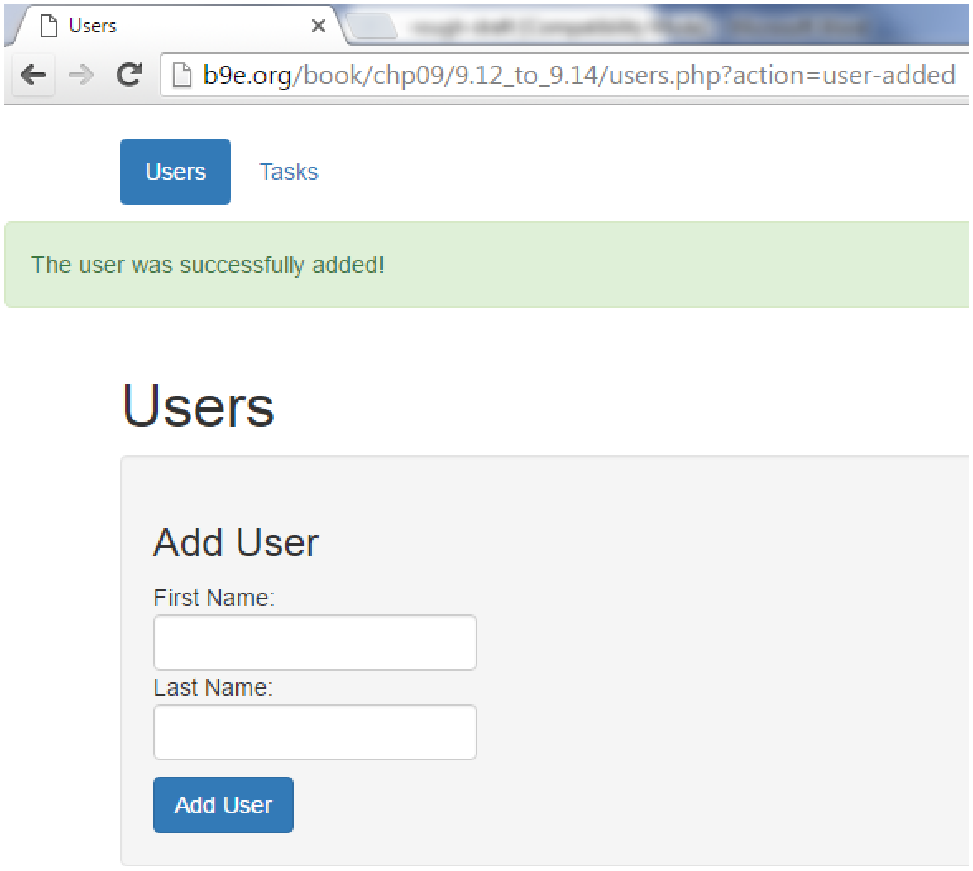
We need to include this code in Template_Header() function in layouts.php to match Figure 9.15.

layouts.php

```
<?php function Template_Header() { ?>

 ... HTML/HEAD code here ...

<body>
      <div class="container" style="margin-top:20px;">
            <ul class="nav nav-pills">
              <li role="presentation" class="active">
                        <a href="users.php">Users</a>
              </li>
              <li role="presentation">
                        <a href="tasks.php">Tasks</a>
              </li>
            </ul>
      </div>
      <?php include_once("alerts.php"); ?>
<?php } ?>
<?php function Template_Footer() { ?>
  ...
```

Figure 9.15 – Include notifications in layouts.php

If you refresh the page (the one that has "users.php**?action=user-added**") then you will see the "successful" notification. You will not see the other message because the value of "action" is set to "user-added". Only if the URL had a parameter with a value of "blah" would it have shown up.


Users

b9e.org/book/chp09/9.12_to_9.14/users.php?action=user-added

The user was successfully added!

Users

Add User

First Name:

Last Name:

Figure 9.16 – Displaying "User was successfully added" notification (from Figure 9.14 and 9.15)

Remember when…

In a_first.php the *very first line* of code is **ob_start();** and the very last line of a_last.php is **ob_end_flush();** . The reason for this is, if you try to redirect the user to a new page in the middle of the script (as we're doing here), you'll get an error. These two functions prevent that error.

Let's clean up the controller:

SQL Injection
We are able to get user input (ex. from a textbox) and store it in a database. There is a problem though: we cannot trust the user's data. There is a concept of "SQL Injection," where a user could put in ("inject") malicious values in the input boxes that would alter how the SQL statement was intended to be run. The user could potentially type something in the textbox that, when submitted to the form and the query is run, it could delete the entire database! We need to prevent this.

I primarily use two methods to prevent this:

- If I'm expecting the value entered to be a number, I'll surround the user's input with the built-in PHP function **`intval()`** - this will convert the value to an integer (whole number), and it will default to "0" if it can't detect a number.
- If I'm expecting the value entered to be text, I pass it into a custom function I made, called **`quote_smart()`** (we'll look at this momentarily).

The most hazardous character is a single quote. There is a built-in PHP function called **`mysqli_real_escape_string()`** that will "escape" (avoid/exclude) a single quote, converting " ' " to " \' " – thus making it safe. My personal function uses this and, while I'm at it, will **`trim()`** the value (remove the spaces), and remove any HTML tags that the user put in. The **`mysqli_real_escape_string()`** function needs two parameters: the connection to the database, and the string it is sanitizing. Therefore my **`quote_smart()`** function needs the same two parameters.

The author's personal best practice

I tend to create a separate file for custom-made functions; let's call it "a_functions.php". Then I include this file on every page by "including" it in a_first.php . One of the functions is called **`quote_smart()`** that can help prevent SQL injection:

a_functions.php

```
<?php

function quote_smart($db_connection, $value) {
      if( get_magic_quotes_gpc() )
        $value = stripslashes( $value );

      $value = mysqli_real_escape_string($db_connection, $value );
      $value = strip_tags($value);
      $value = htmlspecialchars($value);
      $value = trim($value);
      return $value;
}

?>
```

Figure 9.17 – custom-made functions in a separate file.
Basically it adds a "\" before single quotes and it will remove HTML tags.

a_first.php

```
<?php
      ob_start();

      ... database connection goes here ...

      include "layouts.php";
      include "a_functions.php";
?>
```

Figure 9.18 – Include the separate file of custom-made functions in every page

The final controller
Once you *know* you won't any SQL errors, you could potentially replace this:

```
      if (mysqli_query($db1, $query)) {
            header("Location: users.php?action=user-added");
            exit();
      } else {
            print ("<strong>ERROR: "
                        . mysqli_error($db1) . "</strong><br />");
      }
```

With this:

```
        $run_query = mysqli_query($db1, $query);

        header("Location: users.php?action=user-added");
        exit();
```

I made a variable arbitrarily named **$run_query** and set it equal to the command that runs the SQL. Then I redirect the user as I normally would.

The updated file is:

controller_users.php

```
<?php include_once("a_first.php");?>

<?php

if (array_key_exists('AddUser', $_POST)) {

        $first_name = quote_smart($db1, $_POST['first_name']);
        $last_name  = quote_smart($db1, $_POST['last_name']);

        $query = "INSERT INTO Users
                    (`id`, `first_name` ,`last_name`)
                    VALUES ('', '$first_name' , '$last_name')";

        $run_query = mysqli_query($db1, $query);

        header("Location: users.php?action=user-added");
        exit();

}

?>

        This is the "User's Controller"

<?php include_once("a_last.php");?>
```

Figure 9.19 – Updated User's Controller