

Creating Database Interaction Tools for the Web

Senior Project Analysis

Megan Peacy

I love the Internet; there's so much you can do with it. Once the government shut down the National Science Foundation in 1995, and the Internet was open for trade, we've never looked back. Business flourishes here, taking a majority of Internet pages for business use. Since high school I have wanted to create the structure of an e-commerce website just for this reason; if I love the Internet, and the Internet is mostly buying and selling, I am going to need to learn how to handle the shopping applications that sites employ. Thanks to the flexibility of the University of Tampa's art department, I finally have that chance.

While implementing a database into a website can be a great way to dynamically generate content, it is also a security risk because all of the information is in one place. Once a hacker gains access, all of the data could be lost in a moment. To combat malicious attacks, the first thing I did when building my e-commerce site was to create the connection in a more secure way by placing the database definitions and the MySQL connection and `'_select_db'` functions in a separate file that is called from the `index.php` container with the `require_once()` function. By including the file in this way, it is telling the browser to check and see if the file has already been included or not, and if so, not to do so again. The best practice would be to place this script at a level above the root folder, but with my hosting provider, GoDaddy, it is not possible to access this level with my standard hosting plan. For this project, I have created a folder titled 'senior' for all of the files and put the connection script one level above this, at the root folder.

– Explain front end structure

After creating the basic structure for how the front end would work, I began creating a content management system for the site administrator to use. Rather than

having to update the database through PHPmyAdmin, information about products and user accounts can be managed by logging into this sub-area of the site. The login form is accessible at 'senior/cms' and only those authorized will be able to continue onto the management forms. In order to set up a secure login, I created a table that included username, password, and email. Upon further evaluation, I decided that it was necessary to add another field to the *user* table to distinguish what kind of access the user had. Using boolean flags, *1* grants permission to access the CMS and *0* grants permission to the customer login area of the front end (this area has not been included at this point).

Once a user enters his/her account information, it is sent to page that contains the script for account verification. As another security measure, the file name does not use any dictionary words in it's name, and uses a combination of letters and numbers. Because it is not contained within the index.php container, our connection script must be included using *require_once()*. We check that the username and password combination exists within the table, but the password has been store using *crypt()* so that if someone does break into the database, they do not know the identical password. Throughout my research for this project, I found conflicting advice on whether or not it is better to use *md5()* or *crypt()* as the password encryption method. From my understand, it is better practice to use *md5()* as it is a strong form of encryption, but in this example, I have used *crypt()* because *md5()* was not returning consistent values, making user verification impossible.

The form also includes a checkbox for the user to elect to have the browser store his/her username for future visits. If it has been checked upon submission, a cookie is set that contains this information and does not expire until 10 years in the future. If the

username and password match what was found in the database, the user is immediately redirected to the main page of the CMS; if not, he/she is immediately sent back to the login screen.

One of the biggest problems during this project was that the session variables for the user login were not set across the site. It seemed that when creating a session variable named *tp_admin*, it was not accessible from any other page than the one it was set on. This was probably because, as stated, the user interactions do not occur on the page with the login script. To remedy this, I had to include path parameter for which this cookie applied. It seemed that it was automatically being set to just the one page when it really needed to be usable throughout the 'senior/cms' folder. This was the solution to the login process until I started to work with the logout.

Originally, my logic was to do the logout procedure in a separate file, just as the login had functioned. After a few failed attempts at the logic, a 'duh' moment occurred when I realized instead of having the link go a page which immediately redirects to the login page, I could eliminate that middleman as long as I also passed it a variable so that the script could check if I was asking to logout of my session. Now, the login page will always check if I am asking to logout, and can do the correct operations to remove that cookie from the browser. It made sense to me to set the session variable for *tp_admin* to a time that has already passed, just as you would with a regular cookie, and after finding a solution that should have killed the session in this way, I was still not destroying the variable. *Unset()* was the function I wanted, which resolved the issues once I figured out that I needed to use `$_GET` to check if the URL parameter I set for logging out, and not `$_POST`, like I had been doing with the form variables.

As a web developer, I have knowledge of the site's structure and a means of modifying it, but the website's administrator will not. The whole point of the CMS is to make this a cloud system, as my networking textbook would tell you; the user does not need to have any knowledge of how things are working behind the presentation. As I built the core functionalities of the CMS, I needed to figure out a way to allow the user to upload product images and how move them from the client's computer to the server. Through the use of an image resizing class found on PHPclasses.org, I found a class that I could include that would help automate my automation process.

In the beginning, class didn't work for uploading images and gave me an error that I was trying to upload an incorrect file type, or MIME type. After trying to adjust this checkpoint in the code, making sure the MIME types were spelled correctly, I had to scrap that class and try a simpler resizer class, removing all of the checks to make sure that the uploader was at least making versions of the image with the specified proportions and was moving the files to the correct locations. The class first gets the image size, which is a check because it would return false if not an image file type, then uses *preg_match()* to check the file extension against a blacklist array that has file types the object cannot be, such as .php and .js which would be malicious scripts. Using *move_uploaded_file()*, the script can then save the image within a directory for the original-size images.

Once the original has been moved into place, I can create a new instance of the resize class that allows me to call those automated functions for creating the resized image copies. All that I need to do is call the *resizeImage()* function, passing in the parameters, such as width, height, and resize type, and *saveImage()* with the location of

where to save including the file name, and then image quality. As a simple security measure, I placed a blank `index.php` file within each directory level so that anyone who tries to visit the image folders at least does not see the complete list of file names.

In the case of the 'Manage Products' form, I had to include a hidden field that held the URL of the current image so that I would know what the current file was, and could move it to the folder titled 'old' in the event that the user wanted to remove an item from the store. Using `copy()` to move the original-size image to the 'old' folder, and `unlink()` to remove the matching images from the 'originals', 'resized', and 'thumbs' folders, the item appears to have been replaced, but is still saved on the server. This is necessary because users often do stupid things, like replace an image he/she shouldn't have and no longer has the file saved to disk.

With the essential components of the content management system complete, the storefront needed to have the shopping cart implemented. There are some classes for this, and plenty of options for purchase, but nothing seemed to fit the needs of this project. It was easiest for me to create my own structure for the shopping cart. A shopping cart works by saving the product preferences that a user has indicated. Through the form on each item page, he/she selects the quantity to buy. After using cookies and session variables in the CMS, first instinct would be to use this method again. However, browsers limit the amount of cookies a website can store to about 20 cookies, with browser limit of 300. This would probably be enough in the case of our store at this point, but what if more and more products came up for sale? How would we handle this information?

This site works a lot with relational array, such as { "product_ number" => "5" } when I am retrieving information from the database, so I thought this would be a way to

save the choices, with one record translating to one unique product number. Logically, this is a good idea; using a *foreach()* loop, the script can determine whether or not that item number is already in the array, adding the specified quantity, or using *array_push()* to create a new row. Problem is that cookies only store strings, not arrays. After much research, I found that *serialize()* was going to allow for an array to be turned into a string, and that *unserialize()* would allow for the string to be reverted into an array.

Not knowing a thing about this function, I went right in trying to pass the array object into the *serialize()* function, but this would return NULL every time. Using *var_dump()*, I was able to verify that the array did have values in it, so that I wasn't passing an NULL object to the function. Research taught me that in order to serialize an object, it is necessary to create a class that included the magic function *__sleep()* which cleans the object and returns an array with the names of all variables of that object that should be serialized. Every time *serialize()* or *unserialize()* is to be used, a new instance of the serializer class must be created. In this instantiation, the object that the class will handle is passed in here, in my case, the array that I want to be serialized. The class I am using for the serialization also includes a *__construct()* method that gets called immediately and automatically. Shown in the code snippet below, the class variable *\$myid* is set to equal the array that has been passed to the class.

```
class serializer {
    public $myid;
    function __construct($id) {
        $this->myid = $id;
    }
    function __sleep() {
```

```
        return array('myid');
    }
}
```

Once the class is instantiated, we can use *serialize()* and *unserialize()* without problem. However, the array that we created with all of the item numbers and quantities is not what is used as the function parameters, as first thought. Instead, the function takes the variable name of the class instantiation as its argument, and *unserialize()* takes the name of the *serialize()* function. To access the unserialized array, the class property we set to equal the array we had created must also be accessed, as shown below.

```
$test = new serializer($myArray);
$serialized = serialize($test);
$unserialized = unserialize($serialized);
var_dump($unserialized->myid);
```

This method of implementation then allowed me to modify the contents of the cookie easily and use the records within the serialized array to visually show what items the user had saved to his/her cart.

Throughout the process of creating this e-commerce website, there have been a lot of concepts and uses of PHP that I have never before used and plenty of issues that I have had to remedy. I have taken the database management class at the University of Tampa, learning the structure of MySQL, but there has not been any chance for me to learn how databases are used on the web. This project threw me into that, and without the short, but influential internship I participated in at Tampa Digital Studios and the only professor on

campus that how to implement this, I wouldn't have had a clue on where to begin writing MySQL statements in PHP. Now I can say that I feel completely comfortable pulling information from a database and populating it into a relational array for easy access. In the future, there is no doubt that I would like to create most if not all of my websites in this way.

The first step in creating the scheduling system for Tampa Digital was to create an HTML form for saving new room reservations. On the 'Reserve' page, users can input the job number, job title, and job description and select the room name from a dropdown menu. Once chosen, the date field's CSS *display* value changes from *none* to *block*, allowing this field to be shown. To allow easy input for the user, the date field uses the jQuery Datepicker plugin in which users can select the date from a visual calendar element. Using jQuery, specific items are binded to specific actions such as click, change, dblclick, and submit which then perform a function when that action occurs. Once the date is selected, an AJAX call goes to the 'scheduled_date' switch case in an external PHP file searches through the 'schedule' table for the previously made reservations for that room on that date. A new, blank array (`$new_array`) is declared to hold the information of these reservations. For each reservation found, a `$time_diff` variable is determined by subtracting the 'start' timestamp from 'end' timestamp and dividing this number by 3600 to find out the difference in minutes. The reservations id, start, end, and length, which is the `$time_diff` value, are pushed into the `$new_array` which is then returned to the browser in a JSON string using `'json_encode($new_array).'` This data is processed within the AJAX call when the status is 'success' AJAX and is used to draw the divisions used to show these previously made reservations. Each of these is then set as jQuery droppable items and, if the user tries to drop the start or end cursor on top of the taken times, the cursor reverts to the (0,0) position. Each of the cursors, for start and end time, are jQuery draggable items, using the 'time_dragger' division as its container. Both snap to a grid of 13 pixels on the x-axis so that it is always adjacent to time interval of 15 minutes (52 pixels equals one hour). However, the end cursor has an additional

constraint: it cannot go to the left of the start cursor's position. By declaring a minimum end position dependent on the x-axis position of the start cursor, the `.draggable()` drag function can test to see if the 'left' attribute of the end cursor is less than that of the start cursor. If it is, a mouseup trigger will be fired, assuring that the end time is never earlier than the start time.

Once the user has selected a time frame for the reservation, meaning they have moved both of the cursors, the person selection is displayed. Again, using an AJAX call, the database is searched to determine what usernames are booked in any and all rooms within the selected time on the selected day. It is not as simple as to getting the records that are within the times; it must also check to see if there are reservations that start before this time and end after the selected start time, start during the selected time and go beyond the selected end time, and reservations that encompass the selected time frame. The 'find_people' switch case determines the start of day (8:00 AM) and end of day (6:00 PM) timestamps to use in the WHERE clauses of these four separate MySQL queries that search through 'schedule'. All of these found rows are then pushed into the blank array \$booked_all. This rows only give us the foreign key for the 'schedule_users' table, so for each record in \$booked_all array, a matching row in the 'schedule_users' table must be found to know what users are a part of each reservation. Another blank array, \$booked_userids, is populated with a record for each user id (1-10) in the 'schedule_users' row that has a 'booking_id' that matches the id of the row in the 'schedule' table. Since the 'schedule_users' table allows for a possibility of reserving up to ten people, the queries are likely to return records with '0' as the value for fields

‘userid2’ through ‘userid10,’ since no user has an id value of 0, this is not data we need in a table consisting of user ids.

Once the \$booked_all array includes the user id numbers of all of the unavailable, we have a list of who is not be included in the people dropdown menu. To collect the names of the people who should, it is necessary to get the names and ids of *all* users and remove the records that correspond with \$booked_userids. To do this, I have used a while loop nested inside of a foreach loop of \$booked_userids, the array that holds the ids of the unavailable users. The while goes until the counter equals the length of the \$all_users array and tests to see if \$all_users[‘i’][‘id’] exactly equals \$booked_userids[‘userid’]. If so, that row of \$all_users is unset and we are left with an array that has the id numbers and usernames, formatted *megan.peacy*. A \$final_array is declared to hold the same information, but changing the formatting of the username to a proper name; *megan.peacy* now becomes *Megan Peacy*. This is the array that gets echoed at the end of the switch case, in the json_encode() string and is used to dynamically create the person dropdowns. There are ten of these dropdowns, corresponding to the ten fields of possible userids in the ‘schedule_users’ table. By default, only on dropdown is shown and subsequent dropdowns can be shown by clicking on the plus sign button to the right of the dropdown. This is again done using .bind() and changing the CSS *display* property from *none* to *block*.

The form fields conclude at this point and the user can click the ‘reserve’ button to save the information. All inputted fields are neatly displayed in divisions so that the information can be verified as correct.

Once reservations have been made, they will show up on the 'overview' page. This is the default page for the schedule section of the web site that displays a black and white outline of the building floor plan. Rooms that cannot be reserved have been greyed out. To the right of this is a dropdown menu that lists all of the dates that have been used for reservations. If the date is today, it is selected; otherwise the dates fall in decreasing order and the date farthest in the future is selected. Below the floor plan is a table that outputs the reservation information. If the user has permissions to edit and delete records, set in the 'access' table, edit and delete icons will appear to the right of the row. I will explain these processes later.

To display the reservations visually on the floor plan image,

Handle access, edit, delete

Created front end

- Creation of database through PHPMyAdmin
- database security

Content Management System

- password encryption
- problem with AJAX bind on elements that are added dynamically
- problems with cookies, user sessions
- image resizer class
- keeping track of items in cart

can only have 20 cookies per site, 300 per browser

had to use `serialize()` to generate a storage value

turns an array into a string

magic function `__sleep()` called before serialization

Scheduling System for TD

- Created form to make a reservation
 - Job number, job title, job description
 - Room drop down
 - Date field using jQuery Datepicker
 - Uses CSS `display:none` changed to `display:block`
- Created tables for 'schedule' and 'schedule_users'
-

Current Categories

- shirt
- print

Add New Category

New Category Name:

Add a New Product

Name:	<input type="text"/>
Picture:	<input type="button" value="Choose File"/> No file chosen
Category:	<input type="text" value="v"/>
Description:	<input type="text"/>
Price:	\$ <input type="text"/>
Units:	<input type="text"/>
<input type="button" value="Save"/>	

WELCOME BACK TPADMIN

Your username has been remembered for future logins.

TEMPUS PROJECTS

PRODUCT MANAGEMENT LOGIN

Username:

Password:

Remember Me

Manage Accounts

Username:	Password:	Email:
<input type="text" value="tpadmin"/>	<input type="password" value="....."/>	<input type="text" value="mpeacy@spartans.ut.edu"/>
<input type="text" value="meganlp"/>	<input type="password" value="....."/>	<input type="text" value="test@test.com"/>

Add New Account

Username:	<input type="text"/>
Password:	<input type="password"/>
Email:	<input type="text"/>

Manage Products

SELECT CATEGORY:

Manage Products

SELECT CATEGORY:

SELECT PRODUCT:

Manage Products

SELECT CATEGORY:

SELECT PRODUCT:

Name:	Picture:	Category:	Description:	Price:	Units:
<input type="text" value="Sagrada Familia"/>	<p>Current Photo:</p>  <p><input type="button" value="Choose File"/> No file chosen</p>	<input type="text" value="print"/>	<input type="text" value="Construction on a cathedral."/>	<input type="text" value="\$ 10.00"/>	<input type="text" value="5000"/>
<input type="button" value="Save Changes"/> <input type="button" value="Delete"/>					

Update Product Units

SELECT CATEGORY:

SELECT PRODUCT:



Name:	Picture:	Category:	Description:	Units:
Musique		print	Celebration images from a Spanish square.	<input type="text" value="5000"/>

MAN NAV

[Print
Shirt](#)

ITEM ADDED. Item 8 now has a total of 3.
[Return to Item page](#)

CART ITEMS:

	Name	Category	Qty	Price	
	Lights	print	Current Qty: 3 New Qty: <input type="text"/>	\$30	<input type="checkbox"/> Remove
	Chanel	print	Current Qty: 1 New Qty: <input type="text"/>	\$10	<input type="checkbox"/> Remove
<input type="button" value="Update"/>				Total: \$40	

MAIN NAV

[Print](#)
[Shirt](#)

Welcome to the storefront for **TEMPUS PROJECTS**

Begin shopping by selecting a product category from the menu at the left.



MAIN NAV

[Print](#)
[Shirt](#)

Lights



Chandelier within a Gothic cathedral of Spain.

Price: \$10.00

Quantity:

1



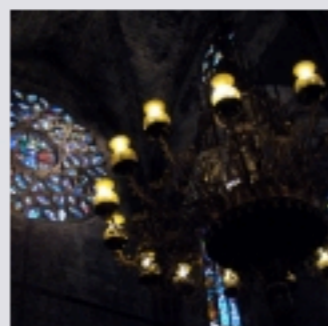
[Add to Cart](#)

TEMPUS PROJECTS

My Cart

MAIN NAV

[Print](#)
[Shirt](#)



[Lights](#)



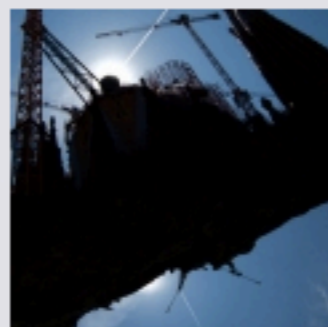
[Chanel](#)



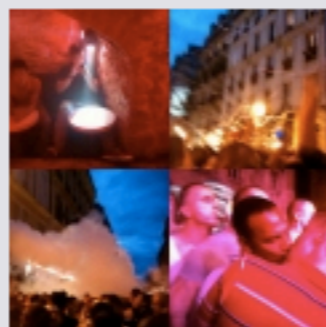
[Eurostar](#)



[Balcony](#)



[Castle in the Sky](#)



[Musique](#)



[Sagrada Familia](#)