Project setup

Create a project database

In this first part, we will create the MySQL database that our project is going to use to store its data. There are three MySQL commands we'll be using. The first is to create the database using **CREATE DATABASE** and then the name of the database. I'm going to call it **chain\_gang**. Second, we want to make sure that we have a user that has the ability to access that database.

CREATE DATABASE chain\_gang

Now, by default, our root MySQL user has access to all of the databases, but it's a much better security practice for us to create a new user that our application can use, instead of using that root user. That root user's just simply too powerful, and can do too much damage to our database. So, we're going to create a new user called **webuser** and give it privileges. So I do that with **GRANT ALL PRIVILEGES ON**, and then I have the database name, followed by a period and an asterisk. That's a shorthand for all tables on the chain\_gang database, and I'm going to assign those to the webuser at localhost. Using localhost ensures that connections can only be made from the local computer, not from some remote IP address somewhere else. And then, IDENTIFIED BY and the password you want to use. I'm using secretpassword. I hope that you'll choose something much better because that's not a very good password. And then, finally, once we've done that, we can actually choose to use our newly created database by typing USE and then the database name, chain\_gang.

GRANT ALL PRIVILEGES ON chain\_gang.\* TO ‘webuser’@’localhost’

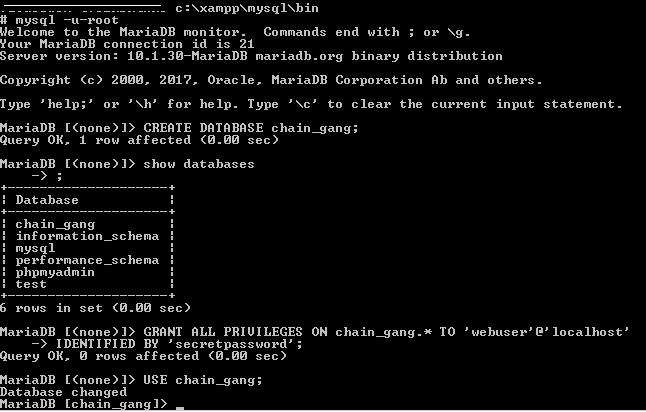
IDENTIFIED BY ‘secretpassword’

USE chain\_gang

Let's try these three out. So, from our command line, I'm going to go into MySQL. I'm going to do that using my root user to begin with. I'm not going to use my web app user to do this; I want my root user that has all of the powers and privileges to create new databases.

**mysql –u root**

So I'll log in with that password, and once I'm inside, then I can CREATE DATABASE chain\_gang. Okay, it says it created the database. We can actually confirm that with SHOW DATABASES. We'll see a few of the basic databases that MySQL needs, but we'll most importantly see that database there, chain\_gang, so we know it was created.



So now, let's grant privileges on it. So we'll do that with GRANT ALL PRIVILEGES, make sure you spell that right, ON chain\_gang.\*, because we want all tables.

And then you can go to a new line here, or you can just keep typing, it doesn't matter. Space, webuser@localhost, and then another space, go ahead and go to a new line here, IDENTIFIED BY and then secretpassword will be mine, but hopefully you'll choose something better. And then the semicolon at the end lets it know that the command is done. Now we have privileges for our webuser, which is great, now we'll be able to log in as the webuser from our PHP code.

And then, last of all, let's type USE chain\_gang. And that will actually switch to the database, so now we can issue all of our commands directly to that database from the command line.

Create a database table

Now that we have a database and we have a user that has the ability to access that database from our application, we're ready to create a database table that's going to store the data about the different bicycles that the chain gang bicycle shop is going to be selling. To do that, we'll need to create a new table, using the SQL syntax. So this is what the table structure is going to look like. We're going to have a number of different attributes, beginning with ID. Every record is going to have an auto incrementing ID, that's going to be its primary key. And then we're going to have columns that correspond to the data that we saw in the table for each bicycle. So we'll have brand, model, year, category, gender, color, price, weight\_kg for kilograms, condition\_ID, and description. Now I know this is a lot of typing. So rather than having to type all of this in, we will make couple of snippets and copy it to command line.

CREATE TABLE bicycles (

id INT(11) AUTO\_INCREMENT PRIMARY KEY,

brand VARCHAR(255) NOT NULL,

model VARCHAR(255) NOT NULL,

year INT(4) NOT NULL,

category VARCHAR(255) NOT NULL,

gender VARCHAR(255) NOT NULL,

color VARCHAR(255) NOT NULL,

price DECIMAL(9,2) NOT NULL,

weight\_kg DECIMAL(9,5) NOT NULL,

condition\_id TINYINT(3) NOT NULL,

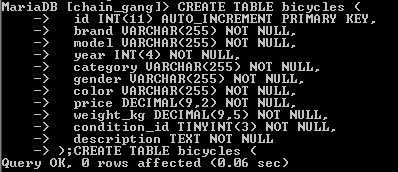
description TEXT NOT NULL

);

And then there's also another one right here, which is insert bicycles.sql, and it has a bit of data that we can insert after we create the tables, so that you'll have a couple of bicycles in the database ready to go. So let's switch to the first one of these. And let's just copy and paste all of that into the My SQL terminal. So I've created my new table bicycles. I can type show tables and I can now see that that table is there. And I can say

SHOW FIELDS FROM BICYCLES

to actually see the list of columns that we have.



Now that we have our table there, let's go ahead and populate it with some initial data. So I'm going to insert some bicycles.

INSERT INTO bicycles (brand, model, year, category, gender, color, price, weight\_kg, condition\_id, description) VALUES ('Trek','Emonda','2017','Hybrid','Unisex','black','1495.00','1.5','5','');

INSERT INTO bicycles (brand, model, year, category, gender, color, price, weight\_kg, condition\_id, description) VALUES ('Cannondale','Synapse','2016','Road','Unisex','matte black','1999.00','1.0','5','');

I'll copy that and let's just go back into the My SQL terminal and paste those both in. So that should have created two bicycles for us, and we can check that those are there using SQL's select statements. Select all from bicycles, and you'll see we get back a long table that just has all the different values that we just input. So now we know that we have two different bicycles in the database. The first one has ID of number one, it's Trek Emonda 2017.

The second one has ID at number two, it's a Cannondale Synapse 2016. You can feel free to put in other data if you want. I just wanted to make sure that we had some initial data in our database so that we can work on reading that data back. Once we've got our database, our table, and some data in it, we're done with My SQL command line for now. You can go ahead and exit out of it, because from now on, we're going to be working with our database from PHP.

The Database Connection

Database credentials

In this chapter we're going to learn how to use object-oriented code to connect and to query the database. If we're going to use a database then our application's going to need to know about our database credentials so that it can log out. Let's start by providing those to our application. So let's start by creating a new file inside the private directory. And I'm going to call that file **db\_credentials.php**.

We'll close them, and let's start by adding a comment here to the top. I'm going to have keep database credentials in a separate file. And the reason why we want to do that is one, it's easy to exclude this file from source code managers. The second reason is because it allows us to have unique credentials on development and production servers.

And also if we're working with multiple developers it allows us to have unique credentials for each of those developers. Misspelled multiple there. There we go. So that's just a reminder why we want to keep these in a separate file. This gives us one file and then we can just exclude that file, everyone can have a unique version of it. Everything else in the application can be identical; it's just this one file that has to be different. Okay, so now let's get about the business of actually setting those values so we can use them.

I'm going to do that by setting some constants. DB\_SERVER is going to be the first one, and I'm just going to be connecting from local host. If we were connecting remotely that would be an IP address. And then define db user, and that's going to be the user that I created that has access to the MySQL table. For me that was webuser. And let's define another one for db pass, which is going to be the password that we used. Mine was secretpassword. Yours hopefully was something different and better.

And then define db name, and that's the name of the database that we want to connect to, which is just chain\_gang. Okay, so there it is. I now have defined these four constants, and then I'll be able to use them globally throughout my application.

// Keep database credentials in a separate file

//1. Easy to exclude this file from source code managers

//2. Unique credentials on development and production server

//3. Unique credentials for multiple developers

define("DB\_SERVER", "localhost");

define("DB USER", "webuser");

define ("DB\_PASS", "secretpassword");

define("DB\_NAME", "chain\_gang");

So let's save that and let's make sure now that that file gets loaded. So to do that we'll need to go to initialize.php. And right here we have require once functions. Let's just add another line, require once, and then db credentials.php.

And that's it; that's all there is to it. Now our application will load up our database credentials and make sure that they're available to us so that we can use them to connect to the database.

Connect to the database

Now that we have our database credentials installed, let's see how we connect to the database using object-oriented programming. First, let's begin by reviewing the procedural version which we've used to connect to the databases with PHP up until now. To do that, we used a function called mysqli\_connect and we passed in several arguments for the server, the user, the password, and the database name. That function returns back a database handle which can then be used to connect to the database in the future.

It opens up a connection and we're able to then work with that connection going forward. So for example if we want to query the database, we can call another function mysqli\_query. We pass in that database handle as well as the SQL that we want to query. This is a procedural style. There's no object-oriented programming here, and we're just calling basic PHP functions.

// Procedural style

$db = mysqli\_connect($server, $user, $pwd, $db\_name);

$result = mysqli\_query($db, $sql);

The mysqli adapter also has an object-oriented version as well as the procedural version. In order to create a new database connection, you create a new mysqli object and you do that by using the new keyword, followed by mysqli, and then passing in those same arguments for server, user, password, and database name. Instead of simply getting back a database handle, it's actually an object and that allows us to work with that database object in an object-oriented way, which means that we can call properties and methods on that object. So for example if we want to perform a query, we can call the query method on that object.

So I have my database connection, I use my arrow notation, followed by the name query for the method, and then in parenthesis the SQL that I want to pass in.

// OOP style

$db = new mysqli($server, $user, $pwd, $db\_name);

$result = $db->query($sql);

Notice the parallel structure between the two of these. They are similar, but there are some important differences. One critical difference is the fact that in the procedural style, we're constantly having to pass this database connection in as one of the arguments. In the object-oriented style, we don't have to do that. The object already knows what its connection is. We've already identified if this is the connection that we're using, we just want to perform a query on that connection.

Let's try adding this to our application. There are a number of different places where we could create this database connection, but I'm going to do it at the bottom of initialize so that way I'll be able to have a database connection from anywhere inside my program. If I call initialize, I'll have that database connection available. So again, we do that by saying new mysqli, that's all lowercase, most of the time when we work with objects in classes they tend to be uppercase, this is actually all lowercase, and then we want to provide those values from our credentials, DB\_SERVER, DB\_USER, DB\_PASS, and DB\_NAME.

$database = new mysqli (DB\_SERVER, DB\_USER, DB\_PASS, DB\_NAME);

Okay, so that will create the connection for me and it will assign it to this variable database. So from then on, I can work with this database. Now, that does it for me, that does the job, but I want to make a slightly improved version of this that allows us to perform some error checking as well so that if for some reason this connection isn't created correctly, we can display an error that the user is able to see. So to do that, I'm actually going to create a new file and I'm going to call it **database\_functions.php.**

And inside that file, I'm going to define a couple of functions. The first one, let's go back over to initialize.php and I'll just copy this as a starting point, database functions, I'm going to create a function called db\_connect and inside there we'll just put that first line. It's just going to connect the database. And I'm going to change this so instead of database, I'm just going to call it connection. And then I'm going to have another function here, this is a procedural function but that's okay, called db\_connect, confirm\_db\_connect, and I'm going to pass in that connection and that's just going to confirm that it's there.

And then if that passes, then we'll just return the connection. Okay, so now we need another function called confirm\_db\_connect which is going to receive a connection as its argument, there we go, and let's add some lines here that are similar to what we did in the PHP with MySQL essential training. We're going to check to see if there is a connection error. So we can do that by using the connection which is now an object so we can just ask for its connect\_errno.

That is a property that is on this object. So if there was an error, that property will have a value. So if it's there, then let's have a message. Our message is going to say database connection failed and then let's say what went wrong as best we can so let's append to our message the connection and we'll use connect\_error, that's another property, this is errno, this is error, that's going to be a text description of the error stored in the object if it had a problem, and I'm just going to copy this line, save me some typing.

This one is going to be the error number and let's also just put some parenthesis around it, space, there we go, and parenthesis at the end so it tells us what the error number was as well, and then let's exit with the message. So that's it. So you'll see here we're already getting the benefits of working with an object because we're able to call its properties for connect\_errno and connect\_error and use those when we're returning it back to the user.

function db\_connect() {

$connection = new mysqli(DB\_SERVER, DB\_USER, DB\_PASS, DB\_NAME);

confirm\_db\_connect($connection);

return($connection);

}

function confirm\_db\_connect($connection){

if($connection->connect\_errno ){

$msg = "Database connection failled";

$msg .= $connection->connect\_error;

$msg .= " ( " . $connection->connect\_errno . " )";

exit($msg);

}

}

Okay, so now that I have these functions created, I need to make sure that initialize knows about them so let's come back over here and let's add a require\_once line for database\_functions.php. And once we have those functions, we don't want to call this anymore. We want to call our new and improved version which is just db\_connect and that's it. So now it's going to call db\_connect, db\_connect is going to do that same step of actually creating a mysqli object, but it's also going to check and make sure that there weren't any errors on the connection in the process before it returns it back to us.

Notice here that I'm mixing both procedural functions with object-oriented programming. That's not a problem. And in fact, you'll find that it's common and often useful.

MySQLi methods and properties

Once we have a database connection object, we can then ask that object for information or to perform tasks for us. After all, that's what object properties and methods are designed to do, to ask for information or to perform tasks. Let's see what some of the methods and properties that are available to us are. The database connection object has many properties and methods, but there are really just four which are most frequently used. The first and certainly the one we use the most often is the one to query the database. The procedural version of this uses mysqli\_query.

The object-oriented version is just simply called query and you're going to see this as a common pattern. What used to be called mysqli\_ is now whatever comes after that underscore as the method name. So here we have the database object, we use the arrow notation, the dash and the greater than sign, and then the name of the method we want to call, which in this case is just query. Another common method we use is mysqli\_real\_escape\_string which is a very long function name.

In the object-oriented version, we have a much shorter option for us. We can just simply call escape\_string. Now real\_escape\_string also works, but why would you type those extra five characters when you can just simply tell it that it's escaping the string? We use this whenever we want to escape a string so that we can use it to construct an SQL query. It helps us to prevent SQL injection attacks. Another one is mysqli\_affected\_rows and that allows us to know how many rows were affected by the last query or last operation.

Unlike the first two, this is not a method on the object, it is a property. Whenever the database connection performs an operation, it automatically stores the number of affected rows from that operation into one of its properties. So it's right there waiting for us and we can tell that it's a property and not a method because it doesn't have parenthesis after it. We have to have a parenthesis if we're talking about a method. We don't have it when we're talking about a property. Another useful property is when we create a new record in the database. We can find out what ID was automatically assigned to it if it was auto incremented.

And we normally would do that using the procedural function mysqli\_insert\_id. But instead of calling a function, we have a property again that automatically is populated with the value from the last created record.

// Procedural style

$db = mysqli\_connect($server, $user, $pwd, $db\_name);

$result = mysqli\_query($db, $sql);

// OOP style

$db = new mysqli($server, $user, $pwd, $db\_name);

$result = $db->query($sql);

$db = mysqli\_connect($server, $user, $pwd, $db\_name);

//db->mysqli\_query($db, $sql); prosedural style

$db->query($sql);

// mysqli\_real\_escape\_string($db, $string);

$db->escape\_string($string);

//mysqli\_affected\_rows($db);

$db->affected\_rows;

//mysqli\_insert\_id($db);

$db->insert\_id;

So these are the four primary methods and properties on the database object. Now if we call a mysqli\_query, we know that we get back a result, right? So the result is what we get back. The result is also an object and it also has properties and methods that we can call on the result object.

So like the database object, there are many of them, but there are really just three that are most frequently used. When we want to get back the values from the result, normally in the procedural version we would use mysqli\_fetch\_assoc for associative array and then provide the result as an argument. Instead, in the object-oriented version, we take the result and we tell it to call its fetch\_assoc method. This will return the first row of data from the result set as an associative array.

Once we are done working through all the different rows in that result that we want, then it's good practice for us to free up the result. So normally we'd use mysqli\_free\_result. The shorter version of that when we're working with object-oriented programming is just result and then call its free method and that will free the result. If we want to know the number of rows that are in the result, we can do that by asking for one of its properties which is num\_rows. Again, this is automatically populated when the result is retrieved so it's a property that's there waiting for us.

Overall, I find that working with the database connection and the database results in an object-oriented way is much simpler and often requires a lot less typing as well. Before we leave this topic and try out some of these methods, I want to note that fetch\_assoc is not the only way to retrieve a row of data. There's also fetch\_row which returns just a basic array instead of an associative array. That's going to be integer indexed instead of being labeled with the column name. And then there's fetch\_array which can return either an associative array, a row, a basic array, or both which is its default.

$result = $db->query($sql);

//mysqli\_fetch\_assoc($result);

$result->fetch\_assoc();

// mysqli\_free\_result($result);

$result->free();

$result->fetch\_assoc(); // associative array

$result->fetch\_row(); //basic array

$result->fetch\_array(); //assoc, row, or both

$result->fetch\_object(); // crude object

And then there's one more which you may have seen before and wondered about which is fetch\_object. Now even though we're using object-oriented programming, I do not recommend using fetch\_object. It does let you work with the results in an object-oriented style, but the objects that it builds are really simple and crude. They're just a set of properties with values and that's rarely useful. As we're going to see throughout the rest of this course, it's much better to get an array of values and then to use that array to build our own custom objects which can have more complex behaviors.

So that's the approach we're going to take.

Query using OOP

Now that we have a database connection, let's try executing an object oriented query over that connection. So the place that we want to do this is inside that public bicycles.php page. You remember that page already lists off the inventory, and right now it's doing that by parsing a CSV file. Right, so it's going and getting that CSV, parsing the data that's in it, and using that to create new bicycles. Eventually what we want to do is replace that with our database driven version. For right now, lets just drop down to the bottom of the page and after our table, lets just have another bit of PHP code that's going to just try and get back values from the database and then we'll just display them below the table.

So we'll be doing both for a little while. We'll be doing the comma separated value parsing and the database. And then eventually we can remove other one when we have this one working. So the first thing we need to do is we need to figure out what our SQL is going to look like. So the SQL is going to be equal to select all from the bicycles table. Okay, so that's simple enough. So now we need to execute that query, that SQL query. We know that we already have the database object set as the variable database, right? That's an initialize.php and all pages are loading that.

You can see the very top here, it's calling initialize.php, and over in initialize.php at the bottom we're creating this variable database which has our database connection. So that means that I can just come down here and I should be able to call on that database for a query of the SQL. And of course that's going to return a results object, so we want to cache that in the variable result, and then we know how to work with that result object in an object oriented way as well. We want to get back the first row of data so we're going to call result and then fetch\_assoc and then that's it, we just call the fetch assoc method and it will return back a result as an array.

I'm going to call that row. It's the first row of data that it's returning and then let's go ahead and tell the result at that point that we're done with it. At which point we can call free, which is a method that will free up the result, and then finally lets take that row and lets just output something so we can take a look at it. Lets do echo brand colon space, and then lets output the value from the row of data and the brand column. So we'll see what the first brand of the first row is.

It'll just echo it back here at the bottom. So that should be enough so that we can at least take a peak and see if we were able to successfully connect to the database, make a query, fetch the row of data from the result object, free it up, and then use those results on our page.

$sql = "SELECT \* FROM bicycles" ;

$result = $database->query($sql);

$row =$result->fetch\_assoc();

$result->free();

echo "BRAND: " . $row['brand'];

Let's go to our browser, and here I am on the same page the chain\_gang/public/bicycles.php, hit return and we get the same comma separated value table. But notice here at the bottom, the important part is the new data that we've added. This is coming from the database right? I've got brand, colon, and then there it is, right? That's the value that's coming back, so that's actually being pulled from the database.

So if we get that, we know that the whole process is working. We know that we created our table successfully and assigned privileges to it, our credentials are good that we provided the application, we're able to create a new My SQL connection object, work within an object oriented way, we get back a result object, work with that in an object oriented way, and then are able to echo back the results to this page. There's a lot more than we can learn, but this is the fundamental process of working with databases in an object oriented way.

Disconnect from the database

We've seen how we can connect to the database, and we've seen how we can use that database connection. Now let's complete the cycle by talking about how we can disconnect from the database. So the way that we create a new database connection is with that new mysqli object command and then we can work with it, when we're finally done we want to close it. In the past you would do that with mysqli\_close and pass it in the database connection. But as we've seen so often before, there's a close method that is on that database object that we can simply call and it will close the connection.

Now this is completely optional. It's not something that you have to do. If you forget to do it, then once your PHP script exits, PHP will close the connection. When that object goes away, it knows that it's time to close the connection. But it is a good practice to go ahead and do it anyway. So we want to be good programmers and follow the best practices. In order to close the connection, once we're done using it, we can simply call database and then close, just like that.

But I think an even better way to do it, to make sure that it's closed all the time is to put it in the footer. You see that all of my pages are going to end with this public footer. So what I'd like to do is just go in to that shared directory, into public footer and down here at the very bottom, then we can just simply put some PHP tags and we'll call database close. Now this would work, however, we would have a problem if for some reason we had a page that was not database enabled. If for some reason we had a page that didn't use the database, and still included this footer, this would raise an error because this variable doesn't exist.

So one improvement that I like to make to this is copy this and let's go back into our database functions. I like to write another function here, that's just called db disconnect. It does the exact same thing. We're going to pass in a connection as a value, and then I'm going to rename this as connection as well. But then I can just do a little error checking. I can just say if is set, connection, then connection close.

And that avoids the error, right. If it's not set, then it won't try and close it. It won't give me an error. If it is set, then it will close it.

function db\_disconnect ($connection){

if(isset($connection)){

$connection->close();

}

}

So now I have this new function I can call instead. I'll just copy that. Let's save it. We'll switch back over to public footer and I'll just change this to be db disconnect and pass in database.

db\_disconnect($database);

So both of them will do the same thing. One is just a little bit more error resistant, so that's the one that I'm going to use. Let's go back to our page, make sure that everything still works, reload the page, and it does.

So our database connection opened. We executed our queries, and then when it loaded the footer, which is down here at the bottom, at that point it closed the database connection as its last task.

Define a Database driven class

Active record design patern

In this chapter, we're going to learn how to define a database driven class. There are a number of different techniques that we could use to do this, but I think that the best one is to use something called the Active Record Design Pattern. The Active Record Design Pattern is a very popular design pattern for using databases with OOP. PHP frameworks like Laravel, CakePHP, Yii, and CodeIgniter all use this Active Record Design Pattern. The idea is that the in-memory objects, that is the instances of a class that we create are going to correspond to a single row in our database table.

The object's properties are going to match the table columns, and the object is also going to have methods to allow it to easily perform the basic database operations of create, read, update, and delete. Let me illustrate. Let's say that we have a table called bicycles that's in our database and it has columns for brand, model, year, et cetera. What we want to be able to do is to retrieve a row from that database so that we have the values that are stored there, and then take those values and automatically populate them into a instance of a class.

So, if we have the bicycle class and it has properties for brand, model, year, et cetera, we would take the values from the database row and use them to create a new instance that has properties with the same values. Because this happens automatically, it feels almost like we're working directly with the database row. The same process also works in reverse. Let's imagine that we create a new instance of the bicycle class, and we assign values for its brand, its model, its year, et cetera. Then, we can take those and we can tell it to save them back to the database.

So, then our in-memory object looks exactly the same as the row in the database. All of those properties in the object get translated into values in a database row. The same thing works with update. You retrieve values out of the database, you make changes to them after they've been created as an object instance, and then you save them back to the database again. Delete works the same way. There's essentially two processes. The first is that we take database records, retrieve them, and their values are used to populate the properties of objects.

Second, an object's properties are used to craft SQL statements that we can submit to the database. In PHP code it looks something like this. Let's say that we create a new instance of a customer object, and then we assign values to several of its properties. At this point, this is just simply a PHP object that has some values assigned to its properties. But the active record portion is that when we call save on that object, it's able to take those properties and convert them into an insert statement that can be submitted to the database all by just calling one single method.

$customer = new Customer;

$customer->name = 'Sally';

$customer->last\_name = 'Jones';

$customer->email = 'sally@jones.com';

$customer->save();

The same thing works in reverse. Let's say that we want to find a customer from the database. We can ask it to retrieve a customer. It would create a new object and automatically populate all of the properties for that object with the values that were stored in the database. We can make changes to them by simply setting new values to the property and calling save again.

$customer = Customer::find\_by\_email('sally@jones.com');

echo $customer->last\_name;

$customer->city = 'Boston';

$customer->save();

It's a powerful, intuitive way for us to work with the rows of data that are in a database. We're first going to implement the Active Record Design Pattern by learning to find records and to create instances on them.

That first process that we talked about. Then in the next chapter, we'll work on creating, updating, and deleting records

Set the database

I want to start implementing the active record design pattern by first telling our object about the database connection that it should be using. In the code that we've already implemented where we're using a database connection, you can see right here that we're calling database query. So this is just simply inside a PHP file. This is not inside a class. What I want to do is I want to be able to move queries like this inside my class, and to do that I need to tell the bicycle class about the database connection. Now I could come over here to my bicycle class and I could do something like use global in order to bring in that variable so that we could use it inside our class.

But that's not ideal for a couple of reasons. It means that every time I have a method that's going to use the database, I have to first remember to tell that method to bring in the database variable from the global scope. A better approach is just to tell this class about its database connection. So we can just simply have a variable called public database and assign a value to it. Now we learned about the difference between instance variables and static variables in object oriented programming. What we want here is a static variable.

That means that it's going to be one value that's going to be shared among all instances of the class. We don't need to have a separate database connection for every single instance. Instead we can just have one, and the class as a whole knows about it, and any of its instances can refer to it whenever it's needed. Now eventually I'm going to make this into a protected method. I'm going to change its visibility from public to protected, but right now we're going to start out with it being public, and I'm going to add another method here, which I'm going to call set database, so function, set database.

We'll pass it in a database, and all it's going to do is set that variable for database equal to whatever we pass in. Now this also needs to be a static function, so I'm going to make it a static public function as well, and I'm also going to just add a little code comment here, just to say that this is the start of active record code, and the reason why is because I want to keep it separate from everything else.

I'm going to keep all of my active record code a little bit separate from now. This is end of active record code. Okay, so now I have this set database that allows me to set it, so now all I need to do is come back over here to initialize, and right here, after I set the database, I just need to tell the bicycle class to set database to database. That's it. Now my bicycle class knows about this database connection.

// ----- START OF ACTIVE RECORD CODE ------

static public $database;

static public function set\_database($database) {

self::$database = $database;

}

// ----- END OF ACTIVE RECORD CODE ------

That's going to be there for it to refer to and for it to use. We can go ahead and start using that since we didn't make it protected. We can actually start using that here in bicycle.php. Let's just change this to be **Bicycle::$database**. That's going to do the same thing. It's going to basically say, hey bicycle class, go to your public database variable, and there you should find a database connection, and then we're going to tell that database connection, perform this query.

$sql = "SELECT \* FROM bicycles";

$result = Bicycle::$database->query($sql);

$row = $result->fetch\_assoc();

$result->free();

echo "BRAND: " . $row['brand'];

Let's try it out just to confirm that it's working. Let's save this, make sure you've saved everything.Let's come back over here and let's reload the page, and sure enough, we get the same thing, brand Trek. It's still working.

Define Query Methods

Now that our bicycle class knows what database connection it should use, we're ready to start defining query methods on the bicycle class. In the previous part, we used that database connection inside the bicycle class to execute a query. What I want to do now is move that query inside the class. The reason why is that it makes sense that a bicycle class ought to know how to retrieve all of the bicycles that are stored in the database. So what we essentially want to do is move these two lines into our class.

So I'm just going to copy those two lines, and what I want to have instead here is I just want to be able to call find all, right? If I just call find all on the bicycle class, it ought to be able to turn back a result to me. Now notice that I'm calling this as a class method, not an instance method. I don't have an instance of bicycle. I haven't found any yet. I'm telling the class that it should use find all.

$result = Bicycle::find\_all();

$row = $result->fetch\_assoc();

$result->free();

echo "BRAND: " . $row['brand'];

All right, so let's save that. Remember, I already copied those lines that we were missing. I want to move them over here, and I will put them also in a static public function called find all, and I'll just paste those in there. Now we do need to make a couple modifications to this. The first is now it's inside a method, so just setting result equals is not good. Instead, we need to actually return back the result. Another change that we should make is that before I had my database property as a public property. I don't need to have that anymore. Before I had this code right here outside the class. So it was necessary for it to be public, but it's better if I can make this protected. Nobody needs to know about this database connection that the bicycle class is going to use except for the bicycle class.

Now, this won't work anymore. I can't access the database this way, but I can use self, and I can access this property still from inside the class using self. So now that we've made those changes, let's save both of those files, and let's go try it out. Come back over here and reload the page, and it still works. Now before we move on from this, there are a couple of improvements that I want to make. The first is that I notice when I'm doing this find all, I'm setting some SQL, then I'm calling query on it.

Now it seems to me that this is the kind of thing that we might want to do in more than one case. This is not the only kind of find we're going to be executing. For example, later we're going to be looking to find a single bicycle in the database. So we're going to be executing more than one kind of query like this. So I'm going to make an all-purpose function public function which I'm going to call find by SQL, and I'll be able to pass SQL into it at all, and it will then execute that SQL. So I'm just going to take this line, I'll copy it, and let's paste it in up here, and now instead of telling the database to query with that SQL, we're going to call that find by SQL function.

So that's all I did. I basically just moved it into its own function, so it should work exactly the same now, but now it's multipurpose. Now I can use this SQL with it, but I could also have another method which had different SQL that would get passed into it. Another improvement that I can make to this then is that I can also put some error checking into find by SQL, because we want to check and make sure that we did get a result, and if we did, we're going to return the result, but in between, let's check and make sure that we actually got a result. So if there's not a result, then let's exit with database query failed.

static public function find\_by\_sql($sql) {

$result = self::$database->query($sql);

if(!$result) {

exit("Database query failed.");

}

return $result;

}

static public function find\_all() {

$sql = "SELECT \* FROM bicycles";

return self::find\_by\_sql($sql);

}

So there we go. So we also have a little bit of error checking in here, and any find by SQL statement that we do is going to execute the query and confirm that it did get back something valid; that the database query didn't fail. Now that we've got that, let's save it all, and one last time, let's go over into Firefox and make sure that everything still works and that we are still able to retrieve this value of Trek from the database.

Build an object using record values

In this movie, we're going to learn to build objects using the values that we pull back from records in the database. This is the first part of the active record process. Database records are retrieved, which we've already done, but then their values are going to be used to populate the properties of objects. And this process needs to happen automatically. That's why it's called active record. It feels like there's an active relationship between the database data and our object. So as soon as we pull those values back, we want to immediately populate a new instance of bicycle with them.

At the moment, we don't have the active record pattern. What we have is a result set, and then we're able to go through and fetch associative arrays out of that result set, which we can then use. Instead, what we want is for find\_all to return an array of objects to us, objects that are already populated with the correct values. So in order to do that, what we want to do is take these two lines, and we want to cut those, because we're going to be moving those over into our class.

$row = $result->fetch\_assoc();

$result->free();

Right here, where we have find\_by\_sql, as soon as we've finished checking that the result is there, before we return the result, what we want to do is turn it into objects. We want to convert the results into objects. I'm going to paste in my two lines now. I'm going to free the result as soon as I am done with it, and what I want to return back is not the result. I want to return back an object array. Okay, so how do we get from fetching the association into the object array? So let's start with an empty object array first.

And then, let's create a loop. So let's say while, let's take this away and let's put parentheses around it, so I'm going to make a while loop here. So while we're able to continue fetching rows out of here, I'm actually going to change this, instead of row, let's make it record. Alright, so while I've gotten a record, then I want to take the object array, and at the end of it, I want to append the record, but I need to instantiate it.

I need to take it and convert its properties. So I'm going to create a new function that'll do that. A new method. It's going to be a static method, so I'm going to use self, and then I'm going to call it instantiate. And I'm going to pass in the record. So this is going to do all the heavy lifting for me. It's going to basically take the values that are in the record, and it's going to return back an object. And then that object will get set to my object array, I can free the result, and then I can return back that object array. So I'll get back an array of objects now, instead.

static public function find\_by\_sql($sql) {

$result = self::$database->query($sql);

if(!$result) {

exit("Database query failed.");

}

// results into objects

$object\_array = [];

while($record = $result->fetch\_assoc()) {

$object\_array[] = self::instantiate($record);

}

$result->free();

return $object\_array;

}

Okay, but what about this instantiate function? We need to write that. So let's drop down here, below find\_all, and let's create static protected function. And I'm going to call it instantiate. It's going to take a record as its argument, and we know that we want to create a new object, in this case new, and it would be bicycle, but we can also just make it self, right? That's going to create a new instance of itself.

Now, I could manually assign the values. Could manually assign values to properties, could do that in a very hand-way, I could go through, and every single one of these properties that are here, for brand, model, year, category, et cetera, I could assign a value to it. But, automatic assignment is going to be faster, call it easier, and reusable. And we're going to be reusing this later.

So I'm going to go ahead and do this automatically. So how can we do that? Well, for each, let's go through the record, right? It's going to be a associative array, so I'm saying go through each pair that's in there. I'm going to call the first one the property, and the second one the value, so loop through there, and then let's check and see, if the property exists, we'll use property\_exists on the object that I'm working with, property, the second value, so if it exists, then let's set its value.

Object, and then dollar sign property, equals dollar sign value. Notice that I'm using a dollar sign here. This is dynamic, right? So whatever that property value is will be here. It'll be brand, the first time, then it'll be model, et cetera, right? That second dollar sign is important. Property, there we go. So that's going to set each one of them. And at the end, we're just going to return back the object. So take a moment to realize how that works.

It's going to loop through all of the different columns that are in the row that I just pulled back, and for every one, it's going to see if that column exists as a property on this object, then assign the value. If not, it'll just get skipped. Now I notice that I am actually missing one value right now, which is public ID. Right, that's one that I added to my database that I did not originally have in my bicycle class. So I'm going to add that as well. So all these properties are here. I also have columns for them, so it should do the work for me.

static protected function instantiate($record) {

$object = new self;

// Could manually assign values to properties

// but automatically assignment is easier and re-usable

foreach($record as $property => $value) {

if(property\_exists($object, $property)) {

$object->$property = $value;

}

}

return $object;

}

So now instantiate should handle the process of converting those values in the row into a new object with properties that have the same values. Let's try all of this out. Let's save it, go back to Bicycles, and now, I'm not getting back a result anymore. Instead, I'm getting back an array of bikes. So now, I should be able to loop through all of those bikes, and output their information. Now I could do that here as a test, but this is actually what I was shooting for the whole time. I've now got an array of the bikes.

So let's cut that, and let's stop doing this test code down at the bottom, and let's actually go up here, and let's replace where we're getting a bike array, right, up here using the parser, let's replace that with our actual bikes. And I don't need to instantiate a new bike anymore here, I can cut that line out, and now, instead, for each of the bikes, as dollar sign bike, now I have an object. See how that works? Take a second to look at it. Find\_all is returning back an array of objects that automatically been populated, with the values that were in the database.

<table id="inventory">

<tr>

<th>Brand</th>

<th>Model</th>

<th>Year</th>

<th>Category</th>

<th>Gender</th>

<th>Color</th>

<th>Weight</th>

<th>Condition</th>

<th>Price</th>

</tr>

<?php

$bikes = Bicycle::find\_all();

?>

<?php foreach($bikes as $bike) { ?>

<tr>

<td><?php echo h($bike->brand); ?></td>

<td><?php echo h($bike->model); ?></td>

<td><?php echo h($bike->year); ?></td>

<td><?php echo h($bike->category); ?></td>

<td><?php echo h($bike->gender); ?></td>

<td><?php echo h($bike->color); ?></td>

<td><?php echo h($bike->weight\_kg()) . ' / ' . h($bike->weight\_lbs()); ?></td>

<td><?php echo h($bike->condition()); ?></td>

<td><?php echo h($bike->price); ?></td>

</tr>

<?php } ?>

And then, I can just loop through those and display them on the page. Let's save this, go back, reload our page, and look at that. Now I'm not getting the values that are coming from that CSV file. Now these are the values that are in my database. That's what I'm seeing instead.

Find a single record

In the previous movie, we retrieved all bicycle records from the database and built new objects whose properties had those same values. In this movie, we're going to learn how to retrieve a single record from the database. This is very common, to want to be able to review all records and then also to be able to focus in on a single record. Let's switch over to our bicycle class and we already have find all. What I want to write now is another method, static public function, and it's going to be called find\_by\_id().

So the idea is that we're going to pass in the ID of the record that we want to it, and it's going to retrieve only that record. So let's start by writing the SQL that we're going to need. So I'll write SQL equals and this is also going to be a select all from bicycles. I'm going to put a space at the end, because I'm going to keep this line going. I'm going to just append it to a new line. And this one is going to be where ID equals and then inside single quotes I'm going to put the ID that we want to retrieve.

Now, that value's going to be dynamic, so I'm going to also put double quotes here, and it's going to be ID like that, and a semicolon at the end. Now we don't want to just take any ID because this ID could be coming from anywhere. We're constructing an SQL statement out of it so what we really want to do is be able to escape the string to prevent SQL injection. We can do that by using the database connection to escape the string. So we know the database connection is stored here. And then, the method that we want there is called escape string and then I'll just put parentheses around it

So that's it. Self colon colon dollar sign database, that's going to be the database connection stored up here. Right, it's stored way up here. And we're going to tell that database connection to call its escape string method passing in that ID as a value. And that'll then get escaped for us. Okay, so that gives me the SQL I can use. So now, what do we want to do? Well, we want to get a result by calling self colon colon find by SQL and pass in our SQL to it, just like we did here with find all.

Now, let's stop and appreciate what happens here. When I call find by SQL, it goes ahead and it instantiates a new object. That is, it takes all of the values that are stored in the database row, and converts them into values for the properties on our object. So we're going to get back an array of objects here as the result. So I'm going to actually change this to be object array, just so that we remember that's what it is. And then let's put in a test here. If it's not empty, then we'll know that we got a value back, right? So if the object array is not empty, then we'll know we have a result.

Else, we didn't find anything. And there's a number of things we could return, an empty value or whatever, but I'm going to return false. Right, so if it returns false if it doesn't find it. If it finds it, it'll return it. Now this is an array, we don't want to just return the object array, that's no good. What we want to actually return is going to be the first item that's in there, right? A good way to do that is with array shift. Array shift'll just take the first item that's in the array and send it back. It basically just pulls the object off the front.

static public function find\_by\_id($id) {

$sql = "SELECT \* FROM bicycles ";

$sql .= "WHERE id='" . self::$database->escape\_string($id) . "'";

$obj\_array = self::find\_by\_sql($sql);

if(!empty($obj\_array)) {

return array\_shift($obj\_array);

} else {

return false;

}

}

So it returns either an object or it returns false, whereas find all is going to return an array of objects. So the behavior'll be just a little bit different. But there's no reason to return an array, if it only has one object inside of it. Now that we've defined our function, in the next movie, we'll create a new PHP page that can make use of it.

Display a detail page

In the last part, we created a new method which would find a record by its ID. I want to build on that and in this part, I want to take that object that we've just instantiated and use it to display a detail page for the bicycle. We need new file called detail.php which we can save into the public directory as detail.php .

<?php require\_once('../private/initialize.php'); ?>

<?php

// Get requested ID

$id = $\_GET['id'] ?? false;

if(!$id) {

redirect\_to('bicycles.php');

}

// Find bicycle using ID

$bike = Bicycle::find\_by\_id($id);

?>

<?php $page\_title = 'Detail: ' . $bike->name(); ?>

<?php include(SHARED\_PATH . '/public\_header.php'); ?>

<div id="main">

<a href="bicycles.php">Back to Inventory</a>

<div id="page">

<div class="detail">

<dl>

<dt>Brand</dt>

<dd><?php echo h($bike->brand); ?></dd>

</dl>

<dl>

<dt>Model</dt>

<dd><?php echo h($bike->model); ?></dd>

</dl>

<dl>

<dt>Year</dt>

<dd><?php echo h($bike->year); ?></dd>

</dl>

<dl>

<dt>Category</dt>

<dd><?php echo h($bike->category); ?></dd>

</dl>

<dl>

<dt>Gender</dt>

<dd><?php echo h($bike->gender); ?></dd>

</dl>

<dl>

<dt>Color</dt>

<dd><?php echo h($bike->color); ?></dd>

</dl>

<dl>

<dt>Weight</dt>

<dd><?php echo h($bike->weight\_kg()) . ' / ' . h($bike->weight\_lbs()); ?></dd>

</dl>

<dl>

<dt>Condition</dt>

<dd><?php echo h($bike->condition()); ?></dd>

</dl>

<dl>

<dt>Price</dt>

<dd><?php echo h(money\_format('$%i', $bike->price)); ?></dd>

</dl>

<dl>

<dt>Description</dt>

<dd><?php echo h($bike->description); ?></dd>

</dl>

</div>

</div>

</div>

<?php include(SHARED\_PATH . '/public\_footer.php'); ?>

It's got a public header. There's a footer at the bottom. It's got a link to go back to the inventory and then it's got code to display the details of a bicycle so we have to have a bicycle object and once we have that, it'll display its properties, the bike brand, the bike model, the year, the category, and so on all the way down the line so it's going to just display all the different properties of the bicycle. We first have to find the bicycle though so that's going to be up here in the top. We've got to fill that in.

Before we do that, let's jump back over here to bicycles.php and let's make a link to that page. So what I'm thinking is that I'll add a new column. In the header I'll just put nbsp.

<th>&nbsp;</th>

Then down here we'll make a new table data row and it's going to say view, all right, but it's going to be a link so we've got to put an href around it and what is that link going to point to? Let's point that to detail.php and then a question mark id equals and then we have to set what the ID is going to be.

We'll do that using php. We're going to echo back the bike object's ID property and that should do it. Let's save it. Let's go to Firefox and take a look. So we've got view here and if you look down in the lower left, you'll see that it has ID equals one and then ID equals two so that's working.

<td><a href="detail.php?id=<?php echo $bike->id; ?>">View</a></td>

And then when we click view, which I'll do right now, it should take us over and show us information about each one of these.

So let's switch back and explain how we got requested ID, and find bicycle using ID in detail.php and l. We're going to need to find a bicycle. So the first thing we need to do is we need to get the requested ID.

We know how to do that with php. Let's do id equals and then we're going to ask for get id and then I'm going to use the PHP 7's null coalescing operator to make it false if for some reason we didn't get an ID and then let's just say if there was no ID, then let's redirect to and redirect to is one of my special functions inside the functions file that allows me to redirect. Let's have bicycles.php.

All right, so that'll just send me back to the index page if I don't have an ID and now we need to find a bicycle so let's have bike equals bicycle colon colon and our new method find\_by\_id and we'll pass in that ID to it and that's it. Now we should have a bike instance. That find by SQL is going to automatically instantiate a new object that has all of the properties set to the same values as the database row for this bicycle. So that'll be set to this.

We'll get back an object. And then down here, we'll display the object brand, the object model, and so on. Let's save it and try it all out. Come back over here and let's reload our page and there you go. I click back to inventory. I can click on the Cannondale bike and I can see the Cannondale's details. See how that works? We're now able to read records back from the database either all of them or a specific record. And in both cases, it's going to have this active record feel because those values that are in the database are automatically going to populate the properties of our objects.

So there's one last thing that I want us to do before we move on. When I click on view, it comes up here and it just says that the page title is called detail. I think it's better if the page title actually had the name of the bicycle here so what I want to have is bike name. Now, I don't have name as an attribute. What I want to do is create a new method over here. Let's go to our bicycle class and down here below our active record code, let's go even below that and let's write a new one a public function called name and this can return anything that you want, but I'm going to make mine return back this brand and then a space curly braces this model and then a space curly braces this year and that's what I'm going to say that the name of each of these is.

So let's go back and let's just reload that page and there we go. Now it says Detail: Track Emonda 2017. Click back to inventory, click on the Cannondale, and that'll just give me a little better page title which shows up in things like your history and your bookmarks and things like that.

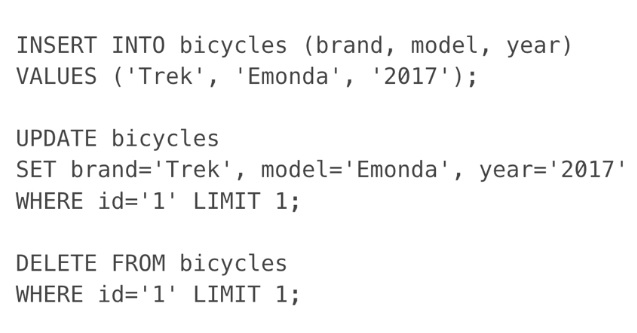
Object-Oriented CRUD

CRUD operations

In this chapter, we'll be learning to perform the remaining database operations so that we can implement the object oriented active record design pattern. We're going to be looking at four main operations, and the abbreviation for these is CRUD. C is for create, R is for read, U is for update, and D is for delete. Those are the four primary ways that we interact with the database. We create new records, we read existing records, we update existing records, or we delete records. So far, we've already seen how we can read from the database when we learned how to do find all and find by ID.

Now we need to learn how to implement the other three. The way that we're going to add that to our project is that we're going to create a staff area, a content management system that'll allow the Chain Gang staff to log in and be able to update the inventory of bicycles using a web interface. They'll be able to create records, update records, delete records, and so on. So the code work that lies ahead of us to implement the active record part of this is we need to be able to take the properties that are an existing object, and we need to be able to transfer those to the database.

So if we have a object that's not save the database, it has properties for brand, model, year, etc., we want to be able to take that and call save on it and have it save to the database, to save a row in the database that matches. Which means we have to translate all those properties into data that can be submitted as an SQL statement. Let's review those SQL statements that we would normally use for these three primary operations.



When we want to create, we use insert. INSERT INTO has a comma separated list of columns followed by a comma separated list of values.

UPDATE uses a different format. Each column is followed by an equal sign and then its value. It also has a WHERE clause, and I'm using a LIMIT 1 as a safety check so that we don't accidentally update more than one record. DELETE just has a WHERE clause so my SQL knows which record we want to delete. Notice that both for DELETE and UPDATE I'm putting the indenture inside single quotes. That's not necessary for SQL, but it is a best practice because it helps prevent SQL injection attacks.

And we are also using LIMIT 1 here so that I don't accidentally delete more than one record. One final point, notice that when I insert a record I am not including its ID. That's because the ID collum uses auto increment so SQL will automatically assign the next available ID when the record is created, And when we want to update the record, we use its ID to find the record, but I do not need to include it in the values that are being updated. So even though ID is a property of the object and it has a collum in the database, I'm going to want to omit that property when I'm assembling the properties that I send to both insert and to update.So let's keep that in mind. ID is a little bit special in that way.

Before we can implement the CRUD, we need to do a little bit of setup in our project. Rather than have you create all of the different HTML files we're going to need, I wanted to give you a jumpstart by giving you some starter code. We will create directory called staff. We will create index.php page and another directory called bicycles, and then we will make files that will allow us to list, show, create, update, and delete bicycle records.

Those are the main operations that we're going to be performing on our bicycles. We have also staff\_ header and a staff\_ footer files. We can take those into the private directory and put them inside shared so they exist in there. And we have got some CSS here, so we can copy that into public style sheets directory so that now we have some special styles for the staff area. And then we have got another set of functions here, status\_error\_functions.php file is here.

So these are missing from our project right now so we can just copy these up into the private directory, and we'll need to go into our project and also go to initialize and make sure that those get added here. So, right after functions we can add line.

require\_once ('status\_error\_functions.php');

So that's this file right here, and you can see it just contains some basic functions for displaying errors, for getting and displaying the session messages back to the user.

Here are pages that we need to create now. First index.php in staff folder:

<?php require\_once('../../private/initialize.php'); ?>

<?php $page\_title = 'Staff Menu'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<div id="main-menu">

<h2>Main Menu</h2>

<ul>

<li><a href="<?php echo url\_for('/staff/bicycles/index.php'); ?>">Bicycles</a></li>

</ul>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

Inside bicycle folder we have files:

staff/bicycles/index.php

<?php require\_once('../../../private/initialize.php'); ?>

<?php

// Find all bicycles;

$bicycles = Bicycle::find\_all();

?>

<?php $page\_title = 'Bicycles'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<div class="bicycles listing">

<h1>Bicycles</h1>

<div class="actions">

<a class="action" href="<?php echo url\_for('/staff/bicycles/new.php'); ?>">Add Bicycle</a>

</div>

<table class="list">

<tr>

<th>ID</th>

<th>Brand</th>

<th>Model</th>

<th>Year</th>

<th>Category</th>

<th>Gender</th>

<th>Color</th>

<th>Price</th>

<th>&nbsp;</th>

<th>&nbsp;</th>

<th>&nbsp;</th>

</tr>

<?php foreach($bicycles as $bicycle) { ?>

<tr>

<td><?php echo h($bicycle->id); ?></td>

<td><?php echo h($bicycle->brand); ?></td>

<td><?php echo h($bicycle->model); ?></td>

<td><?php echo h($bicycle->year); ?></td>

<td><?php echo h($bicycle->category); ?></td>

<td><?php echo h($bicycle->gender); ?></td>

<td><?php echo h($bicycle->color); ?></td>

<td><?php echo h($bicycle->price); ?></td>

<td><a class="action" href="<?php echo url\_for('/staff/bicycles/show.php?id=' . h(u($bicycle->id))); ?>">View</a></td>

<td><a class="action" href="<?php echo url\_for('/staff/bicycles/edit.php?id=' . h(u($bicycle->id))); ?>">Edit</a></td>

<td><a class="action" href="<?php echo url\_for('/staff/bicycles/delete.php?id=' . h(u($bicycle->id))); ?>">Delete</a></td>

</tr>

<?php } ?>

</table>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

staff/bicycles/form\_fields.php

<?php

// prevents this code from being loaded directly in the browser

// or without first setting the necessary object

if(!isset($bicycle)) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

?>

<dl>

<dt>Brand</dt>

<dd><input type="text" name="brand" value="" /></dd>

</dl>

<dl>

<dt>Model</dt>

<dd><input type="text" name="model" value="" /></dd>

</dl>

<dl>

<dt>Year</dt>

<dd>

<select name="year">

<option value=""></option>

<?php $this\_year = idate('Y') ?>

<?php for($year=$this\_year-20; $year <= $this\_year; $year++) { ?>

<option value="<?php echo $year; ?>"><?php echo $year; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Category</dt>

<dd>

<select name="category">

<option value=""></option>

<?php foreach(Bicycle::CATEGORIES as $category) { ?>

<option value="<?php echo $category; ?>"><?php echo $category; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Gender</dt>

<dd>

<select name="gender">

<option value=""></option>

<?php foreach(Bicycle::GENDERS as $gender) { ?>

<option value="<?php echo $gender; ?>"><?php echo $gender; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Color</dt>

<dd><input type="text" name="color" value="" /></dd>

</dl>

<dl>

<dt>Condition</dt>

<dd>

<select name="condition\_id">

<option value=""></option>

<?php foreach(Bicycle::CONDITION\_OPTIONS as $cond\_id => $cond\_name) { ?>

<option value="<?php echo $cond\_id; ?>"><?php echo $cond\_name; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Weight (kg)</dt>

<dd><input type="text" name="weight\_kg" value="" /></dd>

</dl>

<dl>

<dt>Price</dt>

<dd>$ <input type="text" name="price" size="18" value="" /></dd>

</dl>

<dl>

<dt>Description</dt>

<dd><textarea name="description" rows="5" cols="50"></textarea></dd>

</dl>

staff/bicycles/new.php

<?php

require\_once('../../../private/initialize.php');

if(is\_post\_request()) {

// Create record using post parameters

$args = [];

$args['brand'] = $\_POST['brand'] ?? NULL;

$args['model'] = $\_POST['model'] ?? NULL;

$args['year'] = $\_POST['year'] ?? NULL;

$args['category'] = $\_POST['category'] ?? NULL;

$args['color'] = $\_POST['color'] ?? NULL;

$args['gender'] = $\_POST['gender'] ?? NULL;

$args['price'] = $\_POST['price'] ?? NULL;

$args['weight\_kg'] = $\_POST['weight\_kg'] ?? NULL;

$args['condition\_id'] = $\_POST['condition\_id'] ?? NULL;

$args['description'] = $\_POST['description'] ?? NULL;

$bicycle = [];

$result = false;

if($result === true) {

$new\_id = 0;

$\_SESSION['message'] = 'The bicycle was created successfully.';

redirect\_to(url\_for('/staff/bicycles/show.php?id=' . $new\_id));

} else {

// show errors

}

} else {

// display the form

$bicycle = [];

}

?>

<?php $page\_title = 'Create Bicycle'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<a class="back-link" href="<?php echo url\_for('/staff/bicycles/index.php'); ?>">&laquo; Back to List</a>

<div class="bicycle new">

<h1>Create Bicycle</h1>

<?php // echo display\_errors($errors); ?>

<form action="<?php echo url\_for('/staff/bicycles/new.php'); ?>" method="post">

<?php include('form\_fields.php'); ?>

<div id="operations">

<input type="submit" value="Create Bicycle" />

</div>

</form>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

Once that's done, let's try it out, let's take a look. So I'm going to go into browser, and I'm going to change the URL here it's still the public directory but now it's going to go into the staff directory and I can just hit return now or I can type index.php, they both go to the same place. And you'll see I get back my Menu. I now can click on Bicycles. I get a list of bicycles that are in the database. That's using the exact same code that we just learned, let's just go take a look at that real quick.

Inside staff, inside bicycles, inside index.php we're calling find all to get an array of the bicycles, and then down here I'm just looping through that array to display each of the different properties of the bicycle. And then each one of those has a link you can see I've got a link here for View, which will take me to a detail page. Again, that's using those same principles we just saw in the last chapter using find by ID, to find the bicycle. And then I click Back to List, and then I've got pages for Add a Bicycle, which right now is just a blank form, and Edit a Bicycle, which right now is also just a blank form, and Delete a Bicycle.

So those are the three primary pages we're going to be learning how to implement in this chapter. In the chapter movie, let's get started by learning how we create a record.

Create a record

In this part we will learn to create a database record using the properties of an existing PHP object. In order to do that we are going to enable the Add Bicycle page in the content management system we are creating. We already have a form here. We want to be able to fill out this form and click Create Bicycle and have PHP gather up all of the form values that we've just submitted, create and object from it in PHP, and then tell that object that it should save itself to the database.That's that active record design pattern that we've talked about.

Let's go to the code and let's take a look at that new.php page. You can see that's is already got code in here that's going to call initialize.php so we will have a database connection and our bicycle object will know about that database connection, and then we've got code here that says if it's a post request, if we're getting back form parameters, then let's build an associative array called args and it's going to go through each of the post parameters and it's going to set the brand equal to whatever value was submitted in the form for brand, and same thing for model, year, et cetera all the way down the line.

So then we're going to have a set of arguments. Right now I just have a place holder here for bicycle. Bicycle equals an empty array. That's now what we want. What we want is to create a new object from it. We know how to create a new object. We just have new, and then bicycle. So that'll create a new bicycle object. Now what we want to do is have the bicycle object have its properties set to these same values that are in our argument associative array. If you remember, our bicycle class has that ability already. If we scroll down 'til you get to construct, you'll see that when we wrote our construct function in the object oriented programming class, we allowed it to take an argument that is an array.That's called args.

So it's going to take those $args that are passed in and it's going to set the property for brand equal to whatever value is in that associative array. So we have this functionality already built into here. All we have to do is use it. So if we come back over to new.php, put parentheses after bicycle, and pass in that array of args, then that'll automatically be sent to our construct method, and it will set all the properties that we need all the way down the line. Okay, so at that point we now have a PHP object in memory that has the correct properties.

But that doesn't implement the active record pattern that we've been talking about. What we want to do now is take that in memory object and tell it to save itself to the database as a row in the database. And we want to do that by just simply calling bicycle and then call create.

$bicycle->create();

So there's this create method that'll be on bicycle, and it will know how to take all of those properties of the bicycle and turn them into an SQL query that can be submitted. So we've got to write this method called create.

Let's save this, let's switch over to the bicycle class, let's scroll back up to where we've got our active record code, and right here at the end let's create a new public function called create. Now this is an instance function. We have an instance of the bicycle here. We created a new bicycle, so it's an instance method, not a static method. When we were working with find all and find by id we didn't have an instance yet, so we had a static method. This time it's just going to be an instance method, a method on the instance that we've already created.

So how are we going to create this? What does it need to do? What's the process that we need to go through? Well if you think about it we need some SQL, and then we need to send that SQL to the database. So let's have our result equals self database. The bicycle class knows about the database because it has it stored in the database property. And then we can tell that database that it ought to query using our SQL. So if we can construct a good SQL for this, that's all we need to do.

And then last of all, let's just return back the result. The result for an insert operation will either be true or false. So our create will either return true or false, depending on whether or not it succeeded or not. So now how do we write that SQL? Just as a quick review, remember that this is the structure of an SQL insert statement when we want to create something. So we've got insert into and then our table name followed by a comma separated list of the columns. And then after that we have values, and then a comma separated list of the values that we want to put in each of those columns.

public function create() {

$sql = "INSERT INTO bicycles (";

$sql .= "brand, model, year, category, color, gender, price, weight\_kg, condition\_id, description";

$sql .= ") VALUES (";

$sql .= "'" . $this->brand . "', ";

$sql .= "'" . $this->model . "', ";

$sql .= "'" . $this->year . "', ";

$sql .= "'" . $this->category . "', ";

$sql .= "'" . $this->color . "', ";

$sql .= "'" . $this->gender . "', ";

$sql .= "'" . $this->price . "', ";

$sql .= "'" . $this->weight\_kg . "', ";

$sql .= "'" . $this->condition\_id . "', ";

$sql .= "'" . $this->description . "'";

$sql .= ")";

$result = self::$database->query($sql);

if($result) {

$this->id = self::$database->insert\_id;

}

return $result;

}

Let's save that. Let's switch back over to new.php and finish up over here, because we don't want to just create it, we want to also find out what the result was.

$bicycle = new Bicycle($args);

$result = $bicycle->save();

And then it will store a message for the bicycle was created successfully, and it should redirect me back to that detail page. Let's save it and let's try it all out, see if it's working. Let's go back over to our page. Let's first reload it to make sure that everything's still working. It is. And now let's try entering some data. So we've got Schwinn, Cutter, and I'm going to choose 2016 from the list. The category's going to be city. The gender is going to be unisex.

The color will be white. The condition is going to be great. And the weight is going to be 18. And the price is going to be 450. I'm not going to put in a description for now. Let's hit create bicycle. And there you go. It added it, and we know that it added it because it redirected us to that show page which is reading back values from the database. If we click on back to list, you'll see there it is in our list as well. So that does the trick. In the next part let's see how we can improve this a bit.

Dynamic attribute list

In the previous part, we learned to create a new record using an existing PHP object. In this movie, we're going to learn how we can improve that process by creating a dynamic list of the attributes. The part of the code that I want us to focus on is right here where we're listing off all of the different columns that are in the database and then going through and listing by hand each and every one of the properties that we want to assign to those columns. We could do this and it does work, but there are a couple of problems with it.

The first is that I'm eventually going to want to make this code reusable. That's one of our goals later on is to make this code reusable for any database class that we have. It doesn't matter whether it's bicycles, whether it's administrators or customers, we should be able to have some reusable code that's able to just create a record in the database regardless of what columns it might have. So I want to create something that's a little more flexible. The other thing is we're going to be reusing these again when we go to update in our database.

So I'm also thinking about the fact that we're going to have to list all of those out one more time when we create an update. So what I want to do, the improvement that I want to make here is I want to create a list of these attributes to figure out what they are so that we can use them dynamically in our insert statement. So one improvement we can make for that is we can jump up here at the top and right after database, let's add static protected and then db\_columns.

static protected $db\_columns = ['id', 'brand', 'model', 'year', 'category', 'color', 'gender', 'price', 'weight\_kg', 'condition\_id', 'description'];

This is going to be an array of the database columns. We'll have a list of them so that we'll know what they are. We can then loop through that list. So let's list out our columns. A good starting place, we can jump down here to our code, we can copy everything from brand to description.

The bicycle class knows what its database columns are and that's useful and it makes sense that this class ought to have that information. So now that I have that defined, I can use it.

Now, we are going to write a special method that's going to account something for us. It's going to loop through the columns and figure out what all these values are for us. The way that I'm going to do that is by creating another public function called attributes. Attributes is sort of a synonym for properties. It's a little bit more commonly used with the active record pattern than the name properties that PHP uses and it's useful to use something different so that I can tell the difference.

And what I want attributes to be is an array that's an associative array that has both the key and the value so it has both the property name and the value associated with it.

public function attributes() {

$attributes = [];

foreach(self::$db\_columns as $column) {

if($column == 'id') { continue; }

$attributes[$column] = $this->$column;

}

return $attributes;

}

Now, that has to have a dollar sign in front columns of it because it's a dynamic value. It's whatever the name of the column is, right? The first time through, it's going to be brand, model, year, etc. So dollar sign column will do that. And then at the end, I'm also going to have it return back the attributes.

You se that one line here that just says if the column that we're looking at if it's called id, well then we want to skip it, continue. This is because we don’t want id in list of attributes so this is how we will escape it. That's the way we do that. We just skip to the next one so it will not add column. It gets skipped over each and every time. So attributes is going to be, let's make a note here, the properties which have the database columns excluding id.

Okay, so now that we have attributes, we can actually use that. So, up here in create, let's get the attributes and finalise our create method like this:

public function create() {

$attributes = $this->attributes();

$sql = "INSERT INTO bicycles (";

$sql .= join(', ', array\_keys($attributes));

$sql .= ") VALUES ('";

$sql .= join("', '", array\_values($attributes));

$sql .= "')";

$result = self::$database->query($sql);

if($result) {

$this->id = self::$database->insert\_id;

}

return $result;

}

It gives a little more of a parallel structure to it. So the array\_keys from the attributes, the array\_values go here.

So as you can see, this code is much, much simpler. It saves me a lot of typing and it's something that's now a bit more reusable because depending on what the attributes are on the object, it's going to dynamically find them and generate the SQL that it needs.

Now, it's not completely reusable yet. We'll come back to that a little later on. For now, let's just save everything and let's try it out to make sure we didn't break anything. Let's click add bicycle and let's try a new brand. Mongoose, the model is going to be Switchback Sport. The year 2015, there we go. The category will be a mountain bike. The gender will be men's and then the color will be blue.

The condition is going to be decent and the weight will be 24 and the price is going to be 399. Create the bicycle and it worked. Look at that. Click back to list and it shows up now in our list of different bicycles. We now have simpler dynamic code and eventually it's going to pay dividends because it's reusable.

Sanitize values for database

Eagle-eyed observers may have noticed that we have a problem inside our project. That is, that we've not been sanitizing the data before we submit it to the database and our SQL STATE. That's an important step. Let's say, for example, that I had an insert statement that looked something like this.

INSERT INTO bicycles (brand, model, year)

VALUES (‘Faker’, ‘Mike’s Bike’, ‘2017’);

We have got values for brand, model, and year that are Faker, Mike's Bike, and 2017. There's a problem with that second value, Mike's Bike. Notice that Mike's Bike is being delimited on either end by single quotes, but it also contains a single quote inside of it.

SQL is going to think that the value that's being submitted is everything from the first single quote up until the single quote that's between the e and the s. Instead, we need to escape that value, so that SQL will know that it's not a meaningful character, that it's not a delimiter, that it's actually part of the text. We do that by putting a backslash in front of it. That escapes the single quote, and renders it harmless. Now, this is just a simple example, and we certainly would want to take care of this in our application, but there are also a lot worse things that could happen as well.

A hacker could use this technique in order to inject other SQL that we don't intend. It's called an SQL injection attack, and we talk about it in the PHP with MySQL Essential Training. So, I'm not going to go back into it again here. Just know that we need to always escape the values before we submit them to the database. And right now, we're not doing that. We are using a dynamic list of the attributes. So that's where we're getting our values from. So what we need is for these attributes to be sanitized. Now, we could just do that by sanitizing the value here before we add it into the attributes, but I don't want to do that to the attributes, I want to leave the attributes as an unsanitized version, so that we could potentially use it in other contexts as well.

But I am going to create a new method, which is going to sanitize them. I'm going to make that a protected function called sanitized\_attributes(). And what it's going to do is it's going to take the attributes, loop through them, and sanitize the values. So let's create a new array called sanitized, and I'm going to use that array to store these sanitized values, so we'll use a foreach loop, using this attributes as, and we're going to have both their key and their value.

So we'll loop through with both a key and their value, for everything that's inside attributes, and for each one, we'll sanitize it. And we'll add the sanitized value into the sanitized array with the same key. But this time, the value is going to be sanitized. And the way we sanitize is by telling the database connection to call its method escape\_string. So I'll have self::$database, that's my database connection. And then I'll tell it to call escape\_string, and I'll pass it in the value.

Then the value will be escaped, it puts backslashes in front of single spaces, but escape\_string actually does even more than that. It checks for other problems as well. Alright, so that should return the sanitized version back. So now that I have sanitized attributes, I'll just copy that, and instead of working with just the raw attributes, let's work with the sanitized version.

protected function sanitized\_attributes() {

$sanitized = [];

foreach($this->attributes() as $key => $value) {

$sanitized[$key] = self::$database->escape\_string($value);

}

return $sanitized;

}

That's all it takes. So now I'm using the sanitized attributes for getting both my keys and my values for my create statement. Let's save it, make sure everything's still working. So let's switch our browser, and let's try it out, make sure we didn't break anything. So let's click on Add Bicycle. I'm going to add new bicycle that's going to be a Diamondback, model will be Overdrive, and, just so that we have something in here with a single quote, let's make it Bob's Overdrive. So that's going to be the name of it, is Bob's Overdrive, it has a single quote in it, now, that hopefully will not break anything.

We're going to make it a 2016 bike. Category will be Mountain, gender will be Unisex, color will be dark green, condition is going to be just Good, the weight is 23.7, and the price'll be $565. No description. Let's click Create Bicycle, and see if it worked. It did. See, we were able to submit Bob's Overdrive. Click back to list. Alright, it did submit it correctly, it didn't have a problem, because we sanitized the value.

Find a record to update

Now that we've learned how to create records, we're ready to move to the next part of CRUD, which is updating. Updating in the CMS is a two-step process. The first thing we need to do is we need to find a record in the database and get it's current data and then display it for the user in a form that they can edit. That's similar to the process that we've had for reading records so far. We're going to read back a record from the database, turn it into a PHP object, and use it to present a form to the user. And then, when the form is submitted with the user's edits, we want to update the record values that are in the database using an object.

So we've got to have both parts of that process. In this, first part, we're going to be looking at the first one, how we can retrieve data and put it on a form for the user to edit. And then in the next movie, we'll look at how we can actually update it in the database. So what we're going to be working on is edit page. If I click on ID number five and I click edit, we get a form here and what we want is to actually have the existing values populate this form for us. We don't have that right now. Let us first write code for edit.php page.

<?php

require\_once('../../../private/initialize.php');

if(!isset($\_GET['id'])) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

$id = $\_GET['id'];

if(is\_post\_request()) {

// Save record using post parameters

$args = [];

$args['brand'] = $\_POST['brand'] ?? NULL;

$args['model'] = $\_POST['model'] ?? NULL;

$args['year'] = $\_POST['year'] ?? NULL;

$args['category'] = $\_POST['category'] ?? NULL;

$args['color'] = $\_POST['color'] ?? NULL;

$args['gender'] = $\_POST['gender'] ?? NULL;

$args['price'] = $\_POST['price'] ?? NULL;

$args['weight\_kg'] = $\_POST['weight\_kg'] ?? NULL;

$args['condition\_id'] = $\_POST['condition\_id'] ?? NULL;

$args['description'] = $\_POST['description'] ?? NULL;

$bicycle = [];

$result = false;

if($result === true) {

$\_SESSION['message'] = 'The bicycle was updated successfully.';

redirect\_to(url\_for('/staff/bicycles/show.php?id=' . $id));

} else {

// show errors

}

} else {

// display the form

$bicycle = [];

}

?>

<?php $page\_title = 'Edit Bicycle'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<a class="back-link" href="<?php echo url\_for('/staff/bicycles/index.php'); ?>">&laquo; Back to List</a>

<div class="bicycle edit">

<h1>Edit Bicycle</h1>

<?php // echo display\_errors($errors); ?>

<form action="<?php echo url\_for('/staff/bicycles/edit.php?id=' . h(u($id))); ?>" method="post">

<?php include('form\_fields.php'); ?>

<div id="operations">

<input type="submit" value="Edit Bicycle" />

</div>

</form>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

So let's go into our code and let's look at edit.php you can see it's calling initalize.php so it is database enabled and everything.

It's going to make sure that it has an ID being passed in as part of the GET request. And if we look over here you'll see that it is there, ID equals five. So we know that we're working with record number five. And if it doesn't get that it'll redirect but if it does have it, then that's what it's going to set ID equal to. Now we have a bit here that says if it's a post request, we're going to skip over that for now, because we're not dealing with a post request which is when we submit a form. We're dealing with a get request. So let's go down here, it's not a get request, but what do we want to do? We want to display the form.

And that form is going to be able to show the current bicycle.

$bicycle = Bicycle::find\_by\_id($id);

if($bicycle == false) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

So right now this just has an empty place-holder with an array in it. Instead, what we want to do is we want to find the bicycle that's in the database. And we know how to do that. We wrote a method for it: bicycle::find by ID and pass it in the ID that was set at the top of the page. That will find the bicycle. Now there's a chance that it won't find it, that we were given an ID but it wasn't found in the database. So let's add a little error-checking in here. If bicycle is equal to false, that's what find ID will return false if it doesn't find it, if it's false then let's redirect to, we'll use another one of our functions, URL four/staff/bicycles/index.php we'll just go back to the list of bicycles.

Okay, so we should have a bicycle at this point. We were given a valid ID, it found it in the database, we now have that bicycle. But it's not going to show up on the form yet. Let's go down here and take a look, you'll see that it's including a file called form field.php all of the form field that we're using are in that file. And the reason it's in a separate file is so that we can share it between both edit.php and new.php. So that the form fields are the same in both of them. The actual form action is different on both and the form buttons are different, but the fields that are in between are going to be shared.

So, let's go over here and take a look at that. So the first thing it does, it just makes sure that it does have a bicycle object before it even loads up. So I make sure that we're only able to access this file when it's include, right? Someone can't access it directly because this won't be set. And then we need to go down here and for each one of these, we need to have the existing values. Right now those are blank. So let's add that in.

Let's add PHP tags, and inside their going to have echo bicycle and we want it's brand property.And I'll put a semi-colon at the end. And we want to make sure we escape that value to make it safe for HTML. And we have a special function that can help us with that called HTML special charts on it. So let's just copy that. Onto the end of the PHP tags. And for everything that is a input of typed text. Let's add that in there.

<dl>

<dt>Brand</dt>

<dd><input type="text" name="brand" value="<?php echo h($bicycle->brand); ?>" /></dd>

</dl>

So for the value here, it's going to be model. And let's scroll passed those color, let's put one down here. Color. Let's do the same thing for weight\_kg and price. And inside the text area for description we can paste it as well. Description. Alright let's save that and let's just check that out so far. Let's come back and just reload our page. And look at that. Oh, we had a problem here. Bicycle weight kg is not accessible.

That is because, if we take a look at our bicycle class. We made it protected. Let's go ahead and change that real quick. Protected, let's make it public. Okay. We had originally done something different there and condition ID, we're going to make public as well. Those are all going to be public properties. Let's come back here and reload our page. There we go, now we're getting a lot of our data. So we can see the existing data. Our select option though, are still not showing up. So let's go back and revisit those on our form field. Let's scroll up to the top there and for year, the way we do this with select options is in the option, right here before the end of it, if it's selected it should be space selected.

But we only want it to be true if it is the one that has the current value, right. If it matches that property. So we've got to write a little bit of PHP here. PHP, and that's going to say if the bicycles year is going to be equal to the year variable as we're looping through, then we're going to echo back selected. And let's just copy all of that.

<dl>

<dt>Year</dt>

<dd>

<select name="year">

<option value=""></option>

<?php $this\_year = idate('Y') ?>

<?php for($year=$this\_year-20; $year <= $this\_year; $year++) { ?>

<option value="<?php echo $year; ?>" <?php if($bicycle->year == $year) { echo 'selected'; } ?>><?php echo $year; ?></option>

<?php } ?>

</select>

</dd>

</dl>

Save it. Let's go over to browser and let's try it out. I'm not just going to reload the page because I don't want a cashed version. I'm actually going to go up here and hit return in the URL bar. And there it is. Now you see it says year, 2016. Okay so now that we've got that. Let's copy this bit of PHP. And I'll just copy that and let's jump down to category and let's add the same thing here. So I put a space, paste in my PHP, but this time I'm going to check whether the bicycle category is equal to the current $category.

And let's jump down here to gender. Put a space and then my PHP code. And lets do gender going to be equal to current gender that I'm looping through and last of all condition. So right here put a space, and then I'll drop in my code. And then let's check for a condition\_ID and then the variable I'm using as I'm looping though is cond\_id. So there we go.

Let's go back over and reload our page and see if it's all working. Again let's don't get a cash version let's go up here and hit return and there it is. Now you see I have 2016, mountain, unisex and good. It did fill in all the correct values for them. Now that I'm able to display all those values, let's just go back, and let's click back to list and let's click add a bicycle. Because I want to show you that we now have a problem. Right? Because our form is being reused in two places and it now expects to have a bicycle object in all cases. Let's go back to new.php and see what's going on there.

So we handled when it's a post request before, that's when we're actually creating a new record. But under the get request, we actually don't have a bicycle here. So what kind of bicycle should it be? Well just any bicycle will do. We just need to have a basic bare bones bicycle when we're creating a new one to start out with. Let's go back over, let's reload our page. And now we don't get an error. Right now it works just as we would expect. We even get condition good here, which if you remember in bicycle class, in construct, that's the default condition right? It defaults to three if no other condition has been passed in.

So it's even allowing us to set default values inside our constructor, and to display those on the form as well. Which is kind of nice. So now that we have our form working, so that we can view the information. Now the user has the opportunity to make changes to it. And then, when they're ready, they can submit the button at the bottom and we'll need to take that information and update the database with it. Let's see how to do that in the next part.

Update a record

In this part, we're going to finish learning to update a record, using object oriented programming and our active record design pattern. So far, we've completed the first part of the update process, which is that we've retrieved a row of data from the database, and turned it into an object, so that we can present it as a form for the user, so the user can edit its properties. Now we need to do the second part, which is we need to take the values that the user submits, and we need to update a row in the database, using SQL.

So, here's the form that we created in the last part, what we want now is when we click Edit Bicycle, and it submits a post\_request, we want it to update the values in the database. So let's go to our code on edit.php, and you can see down here we took care of when we have a get request, that's the else down here, up here is if it's a post\_request, if this form is being submitted. And you can see that we're getting all of the different values that we need from the form. We're putting them into this associative array for args. Now what we need to do is update our bicycle with them.

So the first thing we need to do is get the bicycle again. Right, we need to get the current values out of the database. So that's the same process that we did down here. So I could just take this code and I could copy it and paste it a second time. But that would be repeating myself. If I'm going to always need the bicycle, whether it's a get request or a post\_request, then let's just move it outside of the conditional. I'm going to cut those lines, and then let's just go up here to the top, and above the conditional, let's just find the bicycle. So I get the id and then I find the bicycle object by its id.

$bicycle = Bicycle::find\_by\_id($id);

if($bicycle == false) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

If there is no bicycle, it redirects. Otherwise, we check to see, do we just display the form, or do we update the values using post\_request variables.

How do we update the values inside our object? I have bicycle equals down here. I'm going to take that away. What I want instead, is I want to tell this bicycle that it should merge \_attributes with the arguments. That will update them. Now that's a method on bicycle that doesn't exist yet. We're going to have to write that. But at that point, then, our bicycle object will have the form values, not the database values anymore. And then we can tell our bicycle that it ought to update(). And it will save those values back to the database.

$bicycle->merge\_attributes($args);

$result = $bicycle->update();

So let's write these two methods. First, let's start with merge\_attributes. Save this. We'll switch back over to bicycle class. And right here, after create, I'm going to create another public function. Let's go ahead and create a placeholder for update. We'll come back to that in a second.

But I'm going to make another one then below it, called public function merge\_attributes. And we know that it's going to take args, as its argument. And what do we want to do when we're inside of there? Well, we want to go through each of the arguments that are being passed in, and we want to update the properties for this object accordingly. So for each of the args as key and dollar sign value, let's update them.

If the property exists, good idea to just check and make sure the property is there, the property exists on this instance. Let's check the key to see if the property is there. And if it is, then let's set this key, going to be equal to the value. Notice that I'm using dollar sign key here, because it's a dynamic. I don't actually mean a property called key, I want it to change each time I go through the loop to be brand, model, year, et cetera.

So that's going to have a dynamic property as we go through them. And I'm also going to add one more thing here. Not only am I going to check to see if the property exists, but let's also check and make sure that the value that we're being sent is not null. If it's null, then we won't do anything. If it's false or zero or something else, we will set it, but only in the case when it's null, we're going to ignore it. So if a property didn't show up on the form for some reason, or over here, on edit.php, if we ended up defaulting to null, because it wasn't set for some reason, it's not going to update the attribute.So that means that we'll still get the existing attributes, but we'll just get updates from the form.

public function merge\_attributes($args=[]) {

foreach($args as $key => $value) {

if(property\_exists($this, $key) && !is\_null($value)) {

$this->$key = $value;

}

}

}

All right, so now that we have our attributes merged in, we can now have an object that's in memory, ready to be saved to the database. And we can do that with update. So let's start writing our update. We know that we're going to need some sql for that, and we know that we're going to get back a result from calling self::$database, and telling the database that it should query using that sql. And we're going to return back the result, which is just going to be true or false.

So, now we need to actually write the sql update statement. This is an example of an sql update statement.

UPDATE bicycles

SET brand=’Trek’, model=’Emonda’, year=’2017’

WHERE id=’1’ LIMIT 1;

Notice that it has update, and then the table name. Then set followed by the columns that we want to update and each one of them has an equals sign and the value that it should be set to. It's a little different than what we had with create, where we had the columns and the values separated. Here they are one after another. So brand equals trek, model equals Emonda, year equals 2017. And then I've got where, it's going to say which record I want to update.

So it's going to be id equals and then whatever id of the bicycle I'm looking to update is, followed by limit one. So let's try writing that. So let's have update bicycles.

protected function update() {

$attributes = $this->sanitized\_attributes();

$attribute\_pairs = [];

foreach($attributes as $key => $value) {

$attribute\_pairs[] = "{$key}='{$value}'";

}

$sql = "UPDATE bicycles SET ";

$sql .= join(', ', $attribute\_pairs);

$sql .= " WHERE id='" . self::$database->escape\_string($this->id) . "' ";

$sql .= "LIMIT 1";

$result = self::$database->query($sql);

return $result;

}

And then I'm going to put set on this line with a space after it, because on the next line is where I'm going to start putting those pairs. Now for now I'm just going to leave a placeholder here. Let's just call it attribute\_pairs. Those are going to be the pairs that I'm going to come up with. And then I'm going to have more sql after that, which is going to be the where clause.

But I want to make sure also that I have a space before the where clause, after my attribute\_pairs. So I'll put a space at the beginning of it where id equals and then single quotes. And then inside those single quotes, I'm going to want to put the id, and make sure that I escape it. So I'll just break those single quotes up. It's going to be this id, but then I need to make sure that I escape that value. And I do that with self::$database, escape\_string and pass in the id as a value to it.

So that will escape the value. It'll then drop it in an inside double quotes, let's put a space at the end, semicolon and then the last line is just going to be that limit one, which is a good sanity check. All right, so I've got most of it here, but these attribute\_pairs I punted on for now. Now, you could go through and construct that string, step-by-step, but I think a better way to do it is like we did with create, where we used our attributes, and to dynamically come up with this list. So we know that our attributes that we're going to be using are going to be equal to this object's sanitized\_attributes.

That'll give us sanitized versions that we can use. So now that I have those attributes, what I want to come up with is a set of these attribute pairs. I'm just going to copy that, put it up here with the dollar sign in front of it. The attribute\_pairs is going to start out as an empty array, but I'm going to fill it up with the key value pairs. So for each of the attributes that I just found from sanitized\_attributes, as a key and a value, I'm going to loop through and I'm going to add to attribute\_pairs, just append to the end, a string.

And that string is going to have curly braces for the dollar sign key, and then equals and then single quotes, and then inside those single quotes, more curly braces with the value. So that's going to be the string that I need for each one of these. Now that's an array of these. I also need to join them together with commas. So down here, instead of just attribute\_pairs, I'm going to use join. I'll put a dollar sign on attribute\_pairs, and I'm going to join them together with a comma.

So it's going to join together the sets of attribute\_pairs with a comma in between. Okay, so now we've been able to merge in the attributes, and we've been able to call save, and to construct the right sql that we need. That's the active record portion of it. It's what allows us to go directly from the object's properties, to create sql that will update the database values. Let's save it and let's switch back over to edit.php. And let's just make a few changes over here. We're going to need to get the result from that, which is going to be true or false.

And then if it's true, we'll say bicycles updated correctly, if not we're eventually going to show the errors. Let's don't worry about that too much for now. We'll come back and work on the errors a little later on. For now let's save this page, and let's go try it out. So go back to our page. And before we had Bob's Overdrive as the model, let's take out Bob's and just make it Overdrive. Let's click Edit Bicycle. And look at that, it worked. Click Back to List. There it is. Click Edit and let's make sure that our sanitized\_values worked. We put in Bob's Overdrive, again.

Edit Bicycle. It still works. And I'll just hit Back one last time so that we can take Bob's back out. And there we are. So that completes it. Now we're able to both create new records, and we're able to update records. Hopefully, you're able to see the power of wrapping up this record functionality into these methods for create and update. They contain all of the logic necessary to take what's in an object and to put it into the database. And instantiate does the same thing in reverse. It takes the information that's in the database and converts it into an object.

There's one last thing that I want to do here. I want to add another public function which is going to be called save(). So the thing about creating and updating records is that it's really the same process. I'm saving to the database. How are they different? Well, if a record doesn't exist, then we want to call create. If it already exists, we want to call update. So wouldn't it be great if we could tell the difference and know whether a record exists or not? Well, we can, because we know whether or not this record has an id. So a new record will not have an ID yet.

public function save() {

// A new record will not have an ID yet

if(isset($this->id)) {

return $this->update();

} else {

return $this->create();

}

}

So we can check in our code to see whether or not there's an id. We can have if is set, dollar sign this id. And then put an else. Now if the id is set, then we know it's an existing record. So we can have return this update. But if it's not set, then we know it's a new record, and we can return what we get back from the results of this create. See how that works? So now instead of calling create or update, we can simply call save.

And in fact, I'm going to change these to be protected so that they can't be called anymore. And let's go to the two places where we're calling them. And instead of update, let's just call save there. And on new.php, same thing. We'll just simply call save. And it'll sort out whether or not it's a new record or not. We don't need to worry about that and call the right one. We just say save this, and it will save it, whether it's new or not, it will save it to the database. It will essentially sync the two together.

HTML forms for OOP

In this part, we want to show you a technique for working with form data that's very useful when we're working with object-oriented programming. This technique does not use php at all. It really just uses plain HTML, but we think that it's helpful enough that we should cover it here. So far, when we've been creating our form data, we've been creating our form tags, such as input. We have typed text, and then we've been giving it a name, and the name is just the property that we want to be associated with. We have brand, model, year, etc.

<input type="text" name="bicycle[brand]" value="" />

<input type="text" name="bicycle[model]" value="" />

<input type="text" name="bicycle[year]" value="" />

Altogether our code, we have 10 different form elements and they're and not all input tags, but you get the idea. When we submit our form, it's going to submit values for each one of those properties, and then when we go to process the form values, we're going to take each one of those values that's been submitted in the POST data and we're going to create an associative array where we also assign it to a key by the same name.

$args = $\_POST['bicycle'];

So we have our args and we have a key for brand, and we're going to set it equal to the value that was submitted in the form for brand and so on. Then once we have our associative array of args built up, we're going to pass that into the object to create a new bicycle object with those values.

Okay, so that's how we're doing things so far. Now let's talk about how we can improve this. In HTML, we have the ability to name one of our form fields using square bracket notation, and if we do that, what HTML will do is it will submit a series of values as an array. So, name bicycle and then in square brackets brand means that the HTML form will submit bicycle as the value and it will be an array with the key brand, and then the value of whatever the user has submitted.

<?php

// prevents this code from being loaded directly in the browser

// or without first setting the necessary object

if(!isset($bicycle)) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

?>

<dl>

<dt>Brand</dt>

<dd><input type="text" name="bicycle[brand]" value="<?php echo h($bicycle->brand); ?>" /></dd>

</dl>

<dl>

<dt>Model</dt>

<dd><input type="text" name="bicycle[model]" value="<?php echo h($bicycle->model); ?>" /></dd>

</dl>

<dl>

<dt>Year</dt>

<dd>

<select name="bicycle[year]">

<option value=""></option>

<?php $this\_year = idate('Y') ?>

<?php for($year=$this\_year-20; $year <= $this\_year; $year++) { ?>

<option value="<?php echo $year; ?>" <?php if($bicycle->year == $year) { echo 'selected'; } ?>><?php echo $year; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Category</dt>

<dd>

<select name="bicycle[category]">

<option value=""></option>

<?php foreach(Bicycle::CATEGORIES as $category) { ?>

<option value="<?php echo $category; ?>" <?php if($bicycle->category == $category) { echo 'selected'; } ?>><?php echo $category; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Gender</dt>

<dd>

<select name="bicycle[gender]">

<option value=""></option>

<?php foreach(Bicycle::GENDERS as $gender) { ?>

<option value="<?php echo $gender; ?>" <?php if($bicycle->gender == $gender) { echo 'selected'; } ?>><?php echo $gender; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Color</dt>

<dd><input type="text" name="bicycle[color]" value="<?php echo h($bicycle->color); ?>" /></dd>

</dl>

<dl>

<dt>Condition</dt>

<dd>

<select name="bicycle[condition\_id]">

<option value=""></option>

<?php foreach(Bicycle::CONDITION\_OPTIONS as $cond\_id => $cond\_name) { ?>

<option value="<?php echo $cond\_id; ?>" <?php if($bicycle->condition\_id == $cond\_id) { echo 'selected'; } ?>><?php echo $cond\_name; ?></option>

<?php } ?>

</select>

</dd>

</dl>

<dl>

<dt>Weight (kg)</dt>

<dd><input type="text" name="bicycle[weight\_kg]" value="<?php echo h($bicycle->weight\_kg); ?>" /></dd>

</dl>

<dl>

<dt>Price</dt>

<dd>$ <input type="text" name="bicycle[price]" size="18" value="<?php echo h($bicycle->price); ?>" /></dd>

</dl>

<dl>

<dt>Description</dt>

<dd><textarea name="bicycle[description]" rows="5" cols="50"><?php echo h($bicycle->description); ?></textarea></dd>

</dl>

And the same thing for model, year, and so on. So what that means is when we go to process the form, we're looking at an array of values instead of a bunch of single values that we have to go retrieve. So we can just simply go to the post super global and ask for one item: that is bicycle. And what we get back is an associative array. So instead of having to build up that array by going and retrieving all those values one at a time, we're instantly given an array, just because we named our form fields differently.

It's a super useful technique. You can see how much it simplifies things. Now instead of having to go and retrieve 10 different values and assign them to our associative array and build that up, we can just simply ask for one value in the post variables, and we've got our associative array, ready to go, ready to create a new bicycle. Let's try it in our application. So, you'll want to open up the form fields in bicycle because what we're going to be doing is changing the name here. So for each and every one of these, we're just going to put bicycle and then square brackets are going to go around what was the previous name.

I'm just going to copy this so I can paste it a bunch of times, save myself from typing. And I'm going to do it even to the select option here. Doesn't matter what we do it to. There's year. Here's category. Everywhere we have a name, gender. Come down here to color. And condition id. Now if we ever wanted something to not be included in the array, then we could just simply leave it out, but what this will do is that all of these values we're submitting will now all be inside an array of values.

So now let's save it. So that's what it's going to look like. Now before we submit it, let's go back over and let's change the way that we're working with these on a new.php. Instead of doing all of this work right here, we can simply change this to just be args is going to be equal to, and I'll make it post, bicycle. And then we'll put a semicolon. We'll take all of these lines out, and that's all we have to do. It's that simple. In fact, I'll take that line out as well. I'll copy this. Let's save the page.

$args = $\_POST['bicycle'];

We'll go to edit.php and do the same thing. All of this code here, it's just going to be reduced down to this. Just ask for the contents of this bicycle array, an HTML associative array.

$args = $\_POST['bicycle'];

So let's take that and let's go back to our page, and let's try it out. Let's try a new bicycle. This one's going to be a Schwinn. The model is 21 Speed Suburban CS. The year, we'll make it 2015.

The category will be a hybrid. This will be a women's bike. Color is going to be burgundy. The condition, we'll just make it good. And the weight, we'll make it 20 kilograms, and the price will be $299. All right, let's click create bicycle, and look at that. It still worked. We still got our bicycle. The same thing is true when we go to edit. The only difference we made is just the way that HTML is going to send those values to the post variable and the way that we retrieve them. It's a super handy technique.

Validations and errors

Before we leave the topic of creating and updating records in the database, we want us to talk about validations and errors. The fundamental idea is that we want to have control over the quality of data that gets saved to the database. Not all data is necessarily acceptable, and we want to be able to control that so that we have control over what goes in and comes out of our application. One way to do that is with validations. We check to see if data meets our standards before we put it in the database.

And if it doesn't, then we want to know what went wrong, and we want to tell the user about it so that they can fix the problem, and try to submit it again. So that's where validations and errors come in. The basic structure that we've been using for our validations is just that we're going to keep a list of the errors. So to start with, we'll have an array of errors, that's going to be empty, because there are no errors yet, and then we'll perform a series of validations. And validations are just conditional statements where we check to see if something is true or not. And we can really do just about anything in those validations that we want.

We also saw that we can write some functions that can help us to perform common validations. But the end result is the same, either it's true, or it's false. The data either meets that requirement, or it doesn't, and if it doesn't, then we want to simply add some kind of an indication to our errors array to let us know what the problem was. Then, after we've performed all of our validations, we'll be able to look at that errors array and see. If it's empty, then we know that it passed all of its validations, and the data's good to be sent to the database. But if not, then we don't want to save the record. Instead, we want to display errors back to the user. This technique can be used both in procedural programming, or in our object-oriented programming. Let's see how we can add it to our bicycle class.

To begin with, we will write file called validation\_functions.php. We are going to save it into our private directory of our project. File code is following:

<?php

// is\_blank('abcd')

// \* validate data presence

// \* uses trim() so empty spaces don't count

// \* uses === to avoid false positives

// \* better than empty() which considers "0" to be empty

function is\_blank($value) {

return !isset($value) || trim($value) === '';

}

// has\_presence('abcd')

// \* validate data presence

// \* reverse of is\_blank()

// \* I prefer validation names with "has\_"

function has\_presence($value) {

return !is\_blank($value);

}

// has\_length\_greater\_than('abcd', 3)

// \* validate string length

// \* spaces count towards length

// \* use trim() if spaces should not count

function has\_length\_greater\_than($value, $min) {

$length = strlen($value);

return $length > $min;

}

// has\_length\_less\_than('abcd', 5)

// \* validate string length

// \* spaces count towards length

// \* use trim() if spaces should not count

function has\_length\_less\_than($value, $max) {

$length = strlen($value);

return $length < $max;

}

// has\_length\_exactly('abcd', 4)

// \* validate string length

// \* spaces count towards length

// \* use trim() if spaces should not count

function has\_length\_exactly($value, $exact) {

$length = strlen($value);

return $length == $exact;

}

// has\_length('abcd', ['min' => 3, 'max' => 5])

// \* validate string length

// \* combines functions\_greater\_than, \_less\_than, \_exactly

// \* spaces count towards length

// \* use trim() if spaces should not count

function has\_length($value, $options) {

if(isset($options['min']) && !has\_length\_greater\_than($value, $options['min'] - 1)) {

return false;

} elseif(isset($options['max']) && !has\_length\_less\_than($value, $options['max'] + 1)) {

return false;

} elseif(isset($options['exact']) && !has\_length\_exactly($value, $options['exact'])) {

return false;

} else {

return true;

}

}

// has\_inclusion\_of( 5, [1,3,5,7,9] )

// \* validate inclusion in a set

function has\_inclusion\_of($value, $set) {

return in\_array($value, $set);

}

// has\_exclusion\_of( 5, [1,3,5,7,9] )

// \* validate exclusion from a set

function has\_exclusion\_of($value, $set) {

return !in\_array($value, $set);

}

// has\_string('nobody@nowhere.com', '.com')

// \* validate inclusion of character(s)

// \* strpos returns string start position or false

// \* uses !== to prevent position 0 from being considered false

// \* strpos is faster than preg\_match()

function has\_string($value, $required\_string) {

return strpos($value, $required\_string) !== false;

}

// has\_valid\_email\_format('nobody@nowhere.com')

// \* validate correct format for email addresses

// \* format: [chars]@[chars].[2+ letters]

// \* preg\_match is helpful, uses a regular expression

// returns 1 for a match, 0 for no match

// http://php.net/manual/en/function.preg-match.php

function has\_valid\_email\_format($value) {

$email\_regex = '/\A[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\Z/i';

return preg\_match($email\_regex, $value) === 1;

}

// has\_unique\_username('johnqpublic')

// \* Validates uniqueness of admins.username

// \* For new records, provide only the username.

// \* For existing records, provide current ID as second argument

// has\_unique\_username('johnqpublic', 4)

function has\_unique\_username($username, $current\_id="0") {

// Need to re-write for OOP

}

?>

And then we'll go into the project and take a look. It just contains those basic validation functions that we commonly use, whether something is blank, whether it has a presence, whether it has a length greater than a certain amount, whether it has a length less than a certain amount, whether it has a length exactly amount, whether the length is, and you can provide an associative array of options.

And then whether it's included or excluded from an array, whether something has a string inside of it, whether it's a valid email format, and then eventually we'll work on having a unique username. So those are just some basic validation functions, you can write others that you use as well. Let's make sure our application knows about those by switching to initialize.php.

require\_once ('validation\_functions.php');

Okay so now let's switch to the bicycle class. The first thing I want to do is I need to have a way for my bicycle to know what errors are there? Now I could just have a simple array anywhere, but I might as well have the array inside the class, right? How many errors are there on this instance, on this object, what are the problems that this object has? So I'm going to do that by just having a new property here for errors, which is going to be set to an empty array by default. So that's where I'm going to store all the errors on a bicycle instance before I try and save it.

public $errors = [];

So now, before I try to create it, or, before I try to update it, I need to first check and see, is it valid? Run its validations. So I'm going to need to have a new method here, I'm going to call it protected function validate(). It doesn't need to take any argument because it's just going to use the existing instance's properties. And then, right here, in create, let's just call this validate. And then, it'll check and see, after it calls that method, that method should add any errors to the errors array. So we can just check and see. If it's not empty at the end, this errors then we'll know there was a problem, and we can simply just return false. That's it. We won't do anything else, we won't sanitize the attributes, or assemble our SQL, we'll just simply return false right away. We weren't able to create it. Let's take those same two lines and copy it.

protected function create() {

$this->validate();

if(!empty($this->errors)) { return false; }

Let's come down here to update, and we'll do the same thing. First thing I'll do is run the validations and then check to see did the validations result in adding any errors to the error array?

protected function update() {

$this->validate();

if(!empty($this->errors)) { return false; }

Okay so now we've got those both in there, now we need to actually write the validation function. So our validation function can be anything we want, I'm going to keep it really simple for now. And I'm just going to use is blank. So, if is blank, this brand, if the property brand is blank, then we're going to add to our errors array this errors.

And then we're going to use the square bracket to append to the end. And then brand cannot be blank. And I'll do the same thing, I'll just copy that, we'll make another one here. This one is going to be model, and we'll say model cannot be blank. So I'm just going to check and see, make sure that brand and model have values. Now you'd probably want to write validations for all of the different properties to make sure that they all meet the requirements. For now I'm just going to keep it simple and just have these two.

protected function validate() {

$this->errors = [];

if(is\_blank($this->brand)) {

$this->errors[] = "Brand cannot be blank.";

}

if(is\_blank($this->model)) {

$this->errors[] = "Model cannot be blank.";

}

return $this->errors;

}

Now my validations are the same here for both create and update, that's usually true. You could write something fancier that allowed you to have different validations in each case, but most of the time, if we're really checking to see if the data meets a certain set of standards, the standard is the same, whether we're creating or updating, so that's okay. And then, right before this, when I call the validation, it's also a good practice to just make sure that errors starts out being empty. So errors is going to be set to an empty array, and that way, if I call it more than once, it'll always reset itself back to an empty array before it runs all its validations, I won't accidentally add the same errors twice.

And then, as a good practice, at the end, I'm just going to return back the value of this errors. I don't have to do that, you can see I'm not even catching a value that this is being returned from, but it's a good idea, just to go ahead and have a return value back out of this validate function if I wanted them. Okay so now that I have my validation in place, let's save it, and let's go try it out. Let's go back to Firefox. Let's try adding a new bicycle. Before we do that, actually, let's make sure we have the ability to display the errors, I forgot that. Let's go to new.php.

And let's scroll down here, and you can see I comment it out before.

<?php echo display\_errors($bicycle->errors); ?>

We're going to un-comment that line right after create bicycle, before we display the form, it's going to show the error. So now the errors are not just in an errors array anymore, now it's going to be inside the object, right, remember up here, we have the bicycle object. So it's going to be bicycle. And then it's public, so we can just ask for its errors.

And I'm just going to copy that line. Then switch over to edit.php . And let's find the spot there where we have the errors.

<?php echo display\_errors($bicycle->errors); ?>

Here it is. Let's past that line in there as well. Okay, so now it should display the errors, let's go back and reload our page. Okay, works, there's no errors right now. Now let's try just submitting it, create bicycle. Nope, it didn't work, please fix the following errors. And it told us what the errors are. Notice that we had two errors, the brand can't be blank, and the model cannot be blank. So now, let's try adding one, and let's fill in each of one of those, let's try a Schwinn. It's going to be a Sanctuary seven-speed.

And the year will be a 2016. Category, it's going to be a Cruiser. The gender is a Women's. It's going to be purple. Condition is good. The weight will be 19.5 kilograms. And the price will be 190. Alright, now, let's click submit. And look at that, it worked this time. Let's go back to list, and let's just try editing it. Let's just try taking away the brand, let's click edit bicycle. Nope, sorry, Brand cannot be blank. We need to fix that.

See how that works? So in both cases, it runs our validations before it tries to save or update the record and if there are problems, it shows them back to the user.

Delete a record

Now that we've talked about how to create, read, and update using the active record pattern, we have one more operation to talk about. And that is delete. Delete is by far the simplest of all the four major operations. We don't have to worry about getting any form data to submit to the database, and we don't have to worry about working with any data that we get back from the database. We just simply tell the database to delete a specific record, and it either does or it doesn't. It's that simple. Let's remind ourselves what an SQL delete statement looks like.

DELETE FROM bicycles

WHERE id = ‘1’ LIMIT 1;

It's also very simple. We just have DELETE FROM, and then the table name, and in this case, bicycles. And then we have to tell it how to find the record or records that it should delete. Most of time we just want to delete a single record, and we're going to use it's primary key, which is its id. So I have a WHERE clause, where id='1'. I'm putting that inside single quotes to help prevent SQL injection. And then I'm also using a LIMIT clause after that just to make sure that I don't accidentally delete more records than I intend. This will insure that this delete is only going to delete one and only one record.

Let's try adding that to our bicycle class. So in the bicycle class, inside our active record code here down at the bottom let's add a new public function for delete(). So the idea here is that I've gone to the database. I've already found this object so I have the object, and I can call a method on it called delete(). And delete() is going to have all the information that it needs to know how to delete this object from the database. So we know that we're going to need to write some SQL. For now I'll just put a placeholder for that.

And then we know we're going to want to run that SQL by telling our database, database that it's going to query using our SQL, And it'll get back a result. And for delete statements the result is just true or false. So let's just turn back that result. It either succeeded or it didn't. So now let's write the SQL. We just saw that in the slide. It's, DELETE FROM bicycles, space, and I'll go to a new line, SQL append WHERE id=, and then inside single quotes, we're going to have the id.

I'm going to put a space; I'm going to go to the next line, SQL, it's going to be, LIMIT 1. Now inside these single quotes I need to put the id for this object. So I'll do that by appending this id inside there. But we don't want to just use it as is, we also want to escape the value before we submit it to the database. We know how to do that with self, dollar sign, database, and call it escape\_string method.

We'll pass in this id as an argument. So that'll sanitize the value, and drop it in. Alright, so now we've written an SQL statement. So when we have an object, we can call the delete method and it will delete the underlying record. Now the object is still going to to be there for a little while at least for the duration of this PHP script. And that can be useful. In fact, I'm going to make a note down here at the bottom that just says, after deleting the instance of the object will still exist, even though the database record does not.

This can be useful as in: let's give a little code example here, I can echo back, let's say I have a user and I want to say the first name of the user, space, was deleted, right. I can use that object still in my PHP code because I still have it's property value, even though there's no underlying database record. But we do have to be careful of that for example, we can't call dollar sign user update, after we just finished calling dollar sign user delete.

public function delete() {

$sql = "DELETE FROM bicycles ";

$sql .= "WHERE id='" . self::$database->escape\_string($this->id) . "' ";

$sql .= "LIMIT 1";

$result = self::$database->query($sql);

return $result;

// After deleting, the instance of the object will still

// exist, even though the database record does not.

// This can be useful, as in:

// echo $user->first\_name . " was deleted.";

// but, for example, we can't call $user->update() after

// calling $user->delete().

}

So we'll have to be careful about that, and make sure that we can use it in limited cases because there is no underlying record. We know that the underlying record is gone now because we just deleted it. But we still have an object in PHP's memory that we can use at least temporarily. Alright, so now that we have our delete function, let's save that and let's go use it.

Lets create delete.php page:

<?php

require\_once('../../../private/initialize.php');

if(!isset($\_GET['id'])) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

$id = $\_GET['id'];

if(is\_post\_request()) {

// Delete bicycle

$\_SESSION['message'] = 'The bicycle was deleted successfully.';

redirect\_to(url\_for('/staff/bicycles/index.php'));

} else {

// Display form

}

?>

<?php $page\_title = 'Delete Bicycle'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<a class="back-link" href="<?php echo url\_for('/staff/bicycles/index.php'); ?>">&laquo; Back to List</a>

<div class="bicycle delete">

<h1>Delete Bicycle</h1>

<p>Are you sure you want to delete this bicycle?</p>

<p class="item"><?php echo h('Bike name'); ?></p>

<form action="<?php echo url\_for('/staff/bicycles/delete.php?id=' . h(u($id))); ?>" method="post">

<div id="operations">

<input type="submit" name="commit" value="Delete Bicycle" />

</div>

</form>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

Let's go to delete.php, and to begin with, it's just going to get the id, then if it's a get request, we're going to show a page that it says are you sure you want to delete this bicycle? It's just a confirmation page, and if they post to it and say yes I do want to delete it, then we're we're actually going to execute the delete.

So the delete's going to happen here as a post request. So in either case, we want to to find the bicycle, just like we did when we were updating the record. Let's go to edit.php and here it is right here at the top I'm just going to copy these lines because we ought to be able to find the bicycle to know whether or not we are going to delete it or not. So let's copy those lines and go back to delete.php, there it is.

if(!isset($\_GET['id'])) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

$id = $\_GET['id'];

$bicycle = Bicycle::find\_by\_id($id);

if($bicycle == false) {

redirect\_to(url\_for('/staff/bicycles/index.php'));

}

And right after we find the id, we're going to find the bicycle, now we have an instance, we have an object, cause we're using active record to find it and read from the database.

If we don't find it for some reason, we'll just go back to the index page, but, if we did find it, then let's go down here instead of just bike name, we'll have a placeholder instead we want to say bicycle and let's put in it's name.

<p class="item"><?php echo h($bicycle->name()); ?></p>

That'll just display the name of the bicycle so the user can decide if they want to delete it or not. And then, if they do delete it right here, then we just need to tell that bicycle object to call it's delete method.

Delete. And let's get back it's result, it will return true or false. We could do some error checking here to see if it return false for some reason, we could try to display the errors for why that was false, but that's pretty rare. Most of the time, if we tried to delete it, it's going to delete successfully, so I'm going to leave this and keep it simple and just say it was deleted, and let's redirect them.

$result = $bicycle->delete();

Alright, so let's try it. Let's save this, let's go back, let's start by creating a bicycle that we can destroy. I'm going to make this a junk bike, model is going to be delete me.

The year, I'll just pick something, the category, the gender, color, condition is fine, weight, I'll make it one, and the price will be two. Alright, create bicycle, there it is. Now I've got my junk bicycle, back to list. Here it is, I see the junk bike down here, and I want to delete it, so I click on delete. It opens up the delete page, you can see it's passing in the id of eight. It goes and finds that bicycle, so it has the name. Now we want to click delete bicycle, now it's going to be a post request.

And that post request is going to send that same id, it goes and finds the object, calls the delete method, which sends the SQL, then it redirects us back to the list page. And you can see that the record is gone from the database.

Create inheritable code

We have successfully been able to implement the active record design pattern on our bicycle class. Our bicycle class is able to create, read, update, and delete records in the database. So it allows us to work with those database records in an object oriented way. That's what the active record design pattern is all about. In the next chapter we're going to start working with user authentication. And we're going to want to have a table of the admin users, so the store their username and password, and know who's able to log in. We're also going to want to have a content management system for those, so that we have the ability to create, read updated and delete our admin users.

So we could take all of the code we just worked on, in our bicycle class, we could take it and copy it, and paste it into an admin class, so we'd have that same behavior there. And then if our application gets more robust and the start adding more features, we have more tables, we want to implement it again, we could take that code and copy it and paste it into those classes as while. This violates a fundamental principle of programming. Which is: **don't repeat yourself**. It's a good idea to have code in one place, so that it gets edited only one time, and it helps you to maintain the code, and also keeps you from making careless mistakes.

So a much better approach to this would be to have all of this behavior in one place and allow the bicycle class, the admin class, and any other class that needs it, to be able to use it. We can do that using class inheritance. That's the techniques we talked about in the PHP object oriented programming course. The idea is that we have a parent class, in this case database object, and then we have subclasses that inherit those behaviors from that parent class. Bicycle class, and admin class.

They can both share the behaviors, and we don't have to repeat ourselves. We will have to make some modifications to our code though, because the code in database object has to be a bit more abstract. We have to be able to use it for both the bicycle class, which has a bicycles table, that has its own unique set of columns, and the admin class, which has its own table, and in its own unique set of columns. So we need to have code that's a bit abstract in the parent class so that all the sub classes can use it. Let's add this database object class to our project.

So inside the classes directory let's add a new file, I'm going to call it databaseobject.class.php. And we'll just put PHP tags in there. And class definition for database, object. We'll just make it empty for now. And we'll save that file let's go over to bicycle class. and it is going to inherit from it, so we use the extends key word and then the name of the class database, object. And that's it!

We'll save our file, and now bicycle class is inheriting from the database object class. So now we just need to move the properties and methods that are appropriate into the parent class. Let's go over the parent class and let's think about what some of those might be. To begin with, we know that we're going to want to have a database connection. All of our objects that are going to be database objects are going need to have that. So we'll have static, protected, dollar sign, database. And we know that each one of our classes is going to need to have its table name, right? It's going to need to specify what the table name is.

So I'll have static, protected, dollar sign, table, name. So each one will be able to keep track of it, but to start with it's going to just be empty. And we know that each one of them is also going to have columns protected, dollar sign, columns. Now each one will specify its own, so let's just make it an empty array to start with. And then we know that we're going to want to also have those errors. All of our objects are going to be able to keep track of their errors. So we're going to set that as an initial property as well.

Alright so now let's switch over to the bicycle class. We don't need to specify the database anymore because we just set that there, but we do need to change this to be table name, 'cause we need to specify that the table name is going to be bicycles. Then we're going to need a list of the columns that we're going to have, the errors we don't have to include anymore, and then all of these methods that are down here. Let's just select all of them, I'm going to just like that first line and then scroll down to the bottom. Down where have indirect or code.

And I'm just going to cut all of that. There we go and I can actually remove this start of active record code up here at the top. Let's save it and let's switch over database class, and I'm going to paste all of those in here, every single one of them. Don't need this at the end anymore. Because now this is all active record code. Now it's not abstract enough for us to be able to use yet. But right now this should allow the bicycle class to use it, right, it should still inherit it, we haven't changed anything so it's going to inherit those methods as is and they should still work.

You can try out if you want, but I'm going to set about the task of making it more abstract first. The first thing any do is I need to find all occurrences of the word bicycle, and I instead need to abstract that to be whatever the table name is that's been specified. So let's scroll down 'til we get to find all and you'll see its select all from bicycles. Well its not select all from bicycles anymore, now it needs to append on self colon colon dollar sign table underscore name. That's now going to use that new property, the table name, instead of being a hard coded value, and that will allow this to be inherited.

However, there is an issue. In the object orienting programming course, we talked about the difference between early static bindings and late static bindings. When we use this self keyword, it's going to say that it's bound to the class in which its defined. That is the database object class. It's essentially the same thing as if we had actually typed database object. Instead of using self, what we want to use is static . That's going to determine the class name at run time, so we call find all it's going to see what class is this inside of.

If its inside the bicycle class, then this will be asking for the bicycle table name. If it's the admin class it'll be asking for the admin table name, and that's what we want, so we want to change that to be static. So everywhere we have bicycles, I'm just going to copy this, we're going to put that in there, let's append it together actually, I still need a space at the end. So it's up to you how you add that space, but I do need a space after the table name and before the where clause. So let's make sure that still gets in there, in fact I'm going to copy everything including those double quotes, and let's just scroll down, to the next instance a bicycle, here it is an insert.

I'll drop it in there, and update, I'll paste it in there, and scroll down to delete there we go. And again, just make sure you have that space after the table name. Okay so now I have abstracted the table name, so this would work with any class based on the table name. But this late static binding is actually going to be an issue in other cases as well. There other times in our code where we call self. Let's scroll up to the top, let's click and then I'll do a search for self.

Now here, we're finding self in front of the database. I think that it's actually okay for us to use a single reference for the database. One they gets defined on the database object at the beginning. In fact, I'm going to go over to initialize.PHP. And right here, where we're setting it on the bicycle class, I'm going to change that to sell on the database object class. So it get sat right at the beginning, and then all of my classes that inherit will be using that same property, the one that's defined on the debates object class.

I think that's okay, but every other time we use self, like here, when we call instantiate, we want to be calling it on the static version, the one at run time, right, the sub class. Alright so lets do another find self, find by SQL should also be static. Here's another one here, static, let's do another find. This one's very important and instantiate, we want to make sure that we're getting our static version, so that we're getting the current class, not the database object class.

And that one's okay. That one's okay. Anything that references the database is fine, but anything that reference to something else, we want to change. So I'm going through all of them, lets just a quick check back at the top. Make sure I got 'em all database, database. And yes, I did get 'em all. Okay, I've changed those all so now they use the correct bindings, late static bindings in most cases.

There's one more change that we should make, which is to the validations. Let me just fold these up, until we get to the validate function. validations are going to be specific to each one of our classes. So I'm actually going to take that, and I'm going to copy it, and let's go back to the bicycle class for a moment, and let's add that back in. Now instead of putting it at the top, I'm actually going to drop down and put it the very bottom of the class, so there's my validations, right? So that's going to be unique to each class.

The database object class can just have a generic validation here. Say add custom validations. Alright so we'll override that in the sub class. The sub class will have its own way of handling the validations. If we don't to do any validations, if don't write them, it still does exist, it still does get inherited. Alright, so I think that takes care everything, let's save this, let's go back to the bicycle class. Just going to shoot to the top and make sure I've got everything. The table name, the database columns, the errors is already provided somewhere else, so I'm good.

class DatabaseObject {

static protected $database;

static protected $table\_name = "";

static protected $columns = [];

public $errors = [];

static public function set\_database($database) {

self::$database = $database;

}

static public function find\_by\_sql($sql) {

$result = self::$database->query($sql);

if(!$result) {

exit("Database query failed.");

}

// results into objects

$object\_array = [];

while($record = $result->fetch\_assoc()) {

$object\_array[] = static::instantiate($record);

}

$result->free();

return $object\_array;

}

static protected function instantiate($record) {

$object = new static;

// Could manually assign values to properties

// but automatically assignment is easier and re-usable

foreach($record as $property => $value) {

if(property\_exists($object, $property)) {

$object->$property = $value;

}

}

return $object;

}

static public function find\_all(){

$sql = "SELECT \* FROM " . static::$table\_name;

return static::find\_by\_sql($sql);

}

static public function find\_by\_id($id) {

$sql = "SELECT \* FROM " . static::$table\_name . " ";

$sql .= "WHERE id='" . self::$database->escape\_string($id) . "'";

$obj\_array = static::find\_by\_sql($sql);

if(!empty($obj\_array)) {

return array\_shift($obj\_array);

} else {

return false;

}

}

protected function validate(){

$this->errors = [];

return $this->errors;

}

protected function create() {

$this->validate();

if(!empty($this->errors)){return false;}

$attributes = $this->sanitized\_attributes();

$sql = "INSERT INTO " . static::$table\_name . " (";

$sql .= join(', ', array\_keys($attributes));

$sql .= ") VALUES ('";

$sql .= join("', '", array\_values($attributes));

$sql .= "')";

$result = self::$database->query($sql);

if($result) {

$this->id = self::$database->insert\_id;

}

return $result;

}

protected function update() {

$this->validate();

if(!empty($this->errors)){return false;}

$attributes = $this->sanitized\_attributes();

$attribute\_pairs = [];

foreach($attributes as $key => $value) {

$attribute\_pairs[] = "{$key}='{$value}'";

}

$sql = "UPDATE " . static::$table\_name . " SET ";

$sql .= join(', ', $attribute\_pairs);

$sql .= " WHERE id='" . self::$database->escape\_string($this->id) . "' ";

$sql .= "LIMIT 1";

$result = self::$database->query($sql);

return $result;

}

public function merge\_attributes($args=[]) {

foreach($args as $key => $value) {

if(property\_exists($this, $key) && !is\_null($value)) {

$this->$key = $value;

}

}

}

public function save(){

if (isset($this->id)){

return $this->update();

}else {

return $this->create();

}

}

public function sanitized\_attributes(){

$sanitized = [];

foreach ($this->attributes() as $key => $value){

$sanitized[$key] = self::$database->escape\_string($value);

}

return $sanitized;

}

// Properties which have database columns, excluding ID

public function attributes() {

$attributes = [];

foreach(static::$db\_columns as $column) {

if($column == 'id') { continue; }

$attributes[$column] = $this->$column;

}

return $attributes;

}

public function delete(){

$sql ="DELETE FROM " . static::$table\_name . " ";

$sql .="WHERE id='" . self::$database->escape\_string($this->id) . "' ";

$sql .="LIMIT 1";

$result = self::$database->query($sql);

return $result;

}

}

Bicycle class now has code like this:

class Bicycle extends DatabaseObject {

static protected $table\_name = 'bicycles';

static protected $db\_columns = ['id','brand', 'model', 'year', 'category', 'color', 'gender', 'price', 'weight\_kg', 'condition\_id', 'description'];

// public $errors = [];

public $id;

public $brand;

public $model;

public $year;

public $category;

public $color;

public $description;

public $gender;

public $price;

public $weight\_kg;

public $condition\_id;

const CATEGORIES = ['Road', 'Mountain', 'Hybrid', 'Cruiser', 'City', 'BMX'];

const GENDERS = ['Mens', 'Womens', 'Unisex'];

const CONDITION\_OPTIONS = [

1 => 'Beat up',

2 => 'Decent',

3 => 'Good',

4 => 'Great',

5 => 'Like New'

];

public function \_\_construct($args=[]) {

//$this->brand = isset($args['brand']) ? $args['brand'] : '';

$this->brand = $args['brand'] ?? '';

$this->model = $args['model'] ?? '';

$this->year = $args['year'] ?? '';

$this->category = $args['category'] ?? '';

$this->color = $args['color'] ?? '';

$this->description = $args['description'] ?? '';

$this->gender = $args['gender'] ?? '';

$this->price = $args['price'] ?? 0;

$this->weight\_kg = $args['weight\_kg'] ?? 0.0;

$this->condition\_id = $args['condition\_id'] ?? 3;

//return self::$id++;

// Caution: allows private/protected properties to be set

// foreach($args as $k => $v) {

// if(property\_exists($this, $k)) {

// $this->$k = $v;

// }

// }

}

public function name() {

return "{$this->brand} {$this->model} {$this->year}";

}

public function weight\_kg() {

return number\_format($this->weight\_kg, 2) . ' kg';

}

public function set\_weight\_kg($value) {

$this->weight\_kg = floatval($value);

}

public function weight\_lbs() {

$weight\_lbs = floatval($this->weight\_kg) \* 2.2046226218;

return number\_format($weight\_lbs, 2) . ' lbs';

}

public function set\_weight\_lbs($value) {

$this->weight\_kg = floatval($value) / 2.2046226218;

}

public function condition() {

if($this->condition\_id > 0) {

return self::CONDITION\_OPTIONS[$this->condition\_id];

} else {

return "Unknown";

}

}

protected function validate(){

$this->errors = [];

if(is\_blank($this->brand)){

$this->errors[] = "Brand cannot be blank.";

}

if(is\_blank($this->model)){

$this->errors[] = "Model cannot be blank.";

}

// return $this->errors[];

}

}

Let's go back over make sure we didn't break anything in the process of doing all of that. Go to browser let's add a new bicycle, and I'll call this junk bicycle again. Delete me, and I'll just pick a year. I'll pick a category, pick a gender, pick a color, condition good, weight one, price one create a bicycle. And it worked. Click back to list, and it reads it back just fine. Let's try edit. I'll just change delete me to be capital M.

Edit bicycle, that worked as well. And last of all let's try deleting it. Delete the bicycle. And it deletes it as we would expect. And now we have something that's abstract. We have all of that behavior, wrapped up into this database object class. And any new class can inherit that behavior the same way the bicycle class does. It just calls extends on it, it defines its table name, and its database column, and then if you want validations, you need to define those. That's all it takes for us to be able to have a reusable set of active record behaviors.

Object Oriented Authentication

Admin class

In this chapter, we will add user authentication to the staff area of our project. We will learn to use object-oriented programming techniques to create it. The first step is to create our admin class. To do that, we'll first need to create a database table to store our admins, so I'm going to call that admins and it's going to have a number of columns on it.

Obviously, it will need an ID column for its primary key, but we'll also have first\_name, last\_name, email, username, and hashed\_password as columns. The hashed\_password will store an encrypted version of the password. Once we have database table defined, then we can define our admin class, being sure to inherit behavior from the DatabaseObject class so that we have all of the active record behavior we created in the last chapter. You can refer to the bicycle class if you need guidance on how to do that. And then we'll need to write PHP pages in our staff admins directory that will allow us to manage the admins that are in the database.

That is, to be able to create, read, update, and delete the records that are there. Typically, those pages are called index, show, new, edit, and delete. And a good way to get started on it is to make a copy of the bicycle content management pages that we created and then use find and replace so that it becomes pages for managing admins instead of bicycles.

Okay, let's start by looking at the database table that we'll need. The admins table is going to have a primary key of id. You can see that's an INTEGER size 11, AUTO\_INCREMENT PRIMARY KEY, and then we've got different columns for first\_name, last\_name, email, username, and hashed\_password. All of those are VARCHAR (255) fields. Then you'll notice that after I've created the table, I have another SQL command which is going to add an index to this table.

CREATE TABLE admins (

id INT(11) AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(255) NOT NULL,

last\_name VARCHAR(255) NOT NULL,

email VARCHAR(255) NOT NULL,

username VARCHAR(255) NOT NULL,

hashed\_password VARCHAR(255) NOT NULL

);

ALTER TABLE admins ADD INDEX index\_username (username);

By default, all SQL tables have an index on their primary key and that allows looking up the records by their primary key very quickly. We're also frequently going to be looking up admins by their username. And for any column that we use frequently for look ups, you want to also add an index so that it's also quick to look up those records. So we're going to add an index on the username column with ALTER TABLE admins ADD INDEX and then the name of the index which can be anything we want and then in parenthesis the column that we want to actually index and that's going to be username.

In console we can type SHOW TABLES to see all of them and SHOW FIELDS FROM admins will show me what the admins table looks like. Then I can quit out of mysql because now I have my admins table ready to store my admin records. Now before we go into the code, we also just want to write a couple of files here. We will need admin.class.php which is going to go to in my classes directory and then I've got admins subloder which is designed to go in my staff directory.

Let’s see how the admin class looks like:

class Admin extends DatabaseObject {

static protected $table\_name = "admins";

static protected $db\_columns = ['id', 'first\_name', 'last\_name', 'email', 'username', 'hashed\_password'];

public $id;

public $first\_name;

public $last\_name;

public $email;

public $username;

protected $hashed\_password;

public $password;

public $confirm\_password;

public function \_\_construct($args=[]) {

$this->first\_name = $args['first\_name'] ?? '';

$this->last\_name = $args['last\_name'] ?? '';

$this->email = $args['email'] ?? '';

$this->username = $args['username'] ?? '';

$this->password = $args['password'] ?? '';

$this->confirm\_password = $args['confirm\_password'] ?? '';

}

public function full\_name() {

return $this->first\_name . " " . $this->last\_name;

}

}

You'll notice though that I've made hashed\_password protected so that code outside this class doesn't have access to it and I've created two new ones here which don't have database columns password and confirm\_password. We'll come back to those in the next movie. Then I've got a construct function which is just a convenience so that I can simply pass in an associative array of arguments and it will set those properties equal to the values that get passed in. And then I've also added one for full\_name. That just allows me to have first name and last name together and that's a common method that we use when we have a first\_name and a last\_name column.

Okay, so now that we have those, let's take a look at the admins directory. Admins directory needs index.php.

<?php require\_once('../../../private/initialize.php'); ?>

<?php

// Find all admins

$admins = Admin::find\_all();

?>

<?php $page\_title = 'Admins'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<div class="admins listing">

<h1>Admins</h1>

<div class="actions">

<a class="action" href="<?php echo url\_for('/staff/admins/new.php'); ?>">Add Admin</a>

</div>

<table class="list">

<tr>

<th>ID</th>

<th>First name</th>

<th>Last name</th>

<th>Email</th>

<th>Username</th>

<th>&nbsp;</th>

<th>&nbsp;</th>

<th>&nbsp;</th>

</tr>

<?php foreach($admins as $admin) { ?>

<tr>

<td><?php echo h($admin->id); ?></td>

<td><?php echo h($admin->first\_name); ?></td>

<td><?php echo h($admin->last\_name); ?></td>

<td><?php echo h($admin->email); ?></td>

<td><?php echo h($admin->username); ?></td>

<td><a class="action" href="<?php echo url\_for('/staff/admins/show.php?id=' . h(u($admin->id))); ?>">View</a></td>

<td><a class="action" href="<?php echo url\_for('/staff/admins/edit.php?id=' . h(u($admin->id))); ?>">Edit</a></td>

<td><a class="action" href="<?php echo url\_for('/staff/admins/delete.php?id=' . h(u($admin->id))); ?>">Delete</a></td>

</tr>

<?php } ?>

</table>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

It's going to call admin find\_all, remember it inherited this method so it has it available to us right off the bat. So I've got all my admins. Those are actual objects that have been instantiated and I just need to go through them, so foreach admins as admin and I've got objects and I can call their properties so I can list off the id, the first\_name, the last\_name, and so on. It's not going to display the password of course.

And then I've got links for show, edit, and delete. Let's take a look first at show. Show is going to use find\_by\_id. Again, another one of those methods that we inherited. And then it's going to come down here and just output those values so we can see what a user looks like. So we've got new, if new is a get request, it's just going to create a new blank admin object and display the form. When that form is submitted, then it's going to take those POST parameters, turn them into arguments to create a new admin from, and then tell it to save, again another one of those inherited behaviors.

And edit's going to work in a very similar way. The only difference being that edit is going to always find the existing record in the database before we have the opportunity to edit it. If you scroll down, you'll see that both new and edit are going to make use of form\_fields.php.

<?php

// prevents this code from being loaded directly in the browser

// or without first setting the necessary object

if(!isset($admin)) {

redirect\_to(url\_for('/staff/admins/index.php'));

}

?>

<dl>

<dt>First name</dt>

<dd><input type="text" name="admin[first\_name]" value="<?php echo h($admin->first\_name); ?>" /></dd>

</dl>

<dl>

<dt>Last name</dt>

<dd><input type="text" name="admin[last\_name]" value="<?php echo h($admin->last\_name); ?>" /></dd>

</dl>

<dl>

<dt>Email</dt>

<dd><input type="text" name="admin[email]" value="<?php echo h($admin->email); ?>" /></dd>

</dl>

<dl>

<dt>Username</dt>

<dd><input type="text" name="admin[username]" value="<?php echo h($admin->username); ?>" /></dd>

</dl>

<dl>

<dt>Password</dt>

<dd><input type="password" name="admin[password]" value="" /></dd>

</dl>

<dl>

<dt>Confirm Password</dt>

<dd><input type="password" name="admin[confirm\_password]" value="" /></dd>

</dl>

Let's just open that up and take a look. You'll notice that I'm using that array naming that we talked about so it's admin and then in square brackets first\_name, admin and then in square brackets last\_name, and so on. Basically the same thing we did with bicycle.

And then of course I've got delete.php and delete.php just finds the record and then asks whether we want to delete it or not. And if we post the request, then it will actually call the delete method so that it deletes it. One last thing that we need to do here is let's go into our staff area, the main index.php, this one that's right here, and we need to add a link here to get to our new admin area that we've added in. So I'm just going to copy the line for bicycles, we'll paste it, and instead of going to bicycles, it's going to go to admins and admins.

<?php

require\_once('../../../private/initialize.php');

if(!isset($\_GET['id'])) {

redirect\_to(url\_for('/staff/admins/index.php'));

}

$id = $\_GET['id'];

$admin = Admin::find\_by\_id($id);

if($admin == false) {

redirect\_to(url\_for('/staff/admins/index.php'));

}

if(is\_post\_request()) {

// Delete admin

$result = $admin->delete();

$\_SESSION['message'] = 'The admin was deleted successfully.';

redirect\_to(url\_for('/staff/admins/index.php'));

} else {

// Display form

}

?>

<?php $page\_title = 'Delete Admin'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<a class="back-link" href="<?php echo url\_for('/staff/admins/index.php'); ?>">&laquo; Back to List</a>

<div class="admin delete">

<h1>Delete Admin</h1>

<p>Are you sure you want to delete this admin?</p>

<p class="item"><?php echo h($admin->full\_name()); ?></p>

<form action="<?php echo url\_for('/staff/admins/delete.php?id=' . h(u($id))); ?>" method="post">

<div id="operations">

<input type="submit" name="commit" value="Delete Admin" />

</div>

</form>

</div>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

There we go. Now let's try it on our browser. Go to browser. I'll click on menu. Here's my new admins area. Here I am. It's ready for me to add an admin. I don't have any other admins to list there, but it's ready for us to create an admin. Now, I'm not going to actually go through the process of creating an admin yet because I don't have the ability to hash passwords yet. I need to be able to encrypt passwords and at the moment, that's not a feature I have. So in the next part, let's look at how we do that.

Hashing password

Now that we have the basics of our admin class and the PHP pages that we're going to use to manage it, we're ready to talk about how we hash passwords before we storm in the database and we're going to do it using object-oriented programming. You may remember that PHP offers us a function that makes password hashing very easy. It's very secure and it uses best practices. All we have to do is call password\_hash and pass in the password to it. That's it. As a second argument, we can specify the kind of algorithm that we should use.

By default, that's going to be PASSWORD\_BCRYPT that uses BCRYPT and the blowfish cipher, which is very secure and good for this purpose. So that's all there is to it. We pass in a string to this function and it returns back a hashed version of the password that's suitable for being stored in the database. Now, what I have in mind for the way this is going to work is that we have one property here called password and that's what's going to be set on the form. That's the value that's actually going to be passed into us. It doesn't have a database column, it's just a property on our object.

$this->hashed\_password = password\_hash($this->password, PASSWORD\_BCRYPT);

So let's take a look at admins, here. Look at form fields and you'll see me scroll down here. It's got password, right, so that's what we're going to be passing it is just password. So that value is going to be sent in and, for example, when we create a new object it's going to be in the post parameters, it will become an arg, it will be passed in to the admin object and then, in the admin object, when it constructs it, it's going to set that value, right. So it's going to take the argument password and set it to password. Not hash password.

Hash password is going to be something that's only used internally. This class is going to be able to use it, it's going to be able to encrypt the password and set it, but we're not going to be sending in an encrypted version of the password, right? That's going to happen behind the scenes. So the way that I think we should do that, is let's create a new method here called public function set hash password. So if I call set hash password, it's going to set that value of hash password equal to, and what do we want it to be equal to? We want it to be equal to this password once it's been encrypted.

So we'll call password hash and we'll pass in this password. And we could leave it just like that, but I'm going to go ahead and specify that the algorithm is password bcrypt, just in case that ever changes as the default, I'll have it specified there. So that's it.

protected function set\_hashed\_password(){

$this->hashed\_password = password\_hash($this->password, PASSWORD\_BCRYPT); }

Now I have this public function called set\_hashed\_password. So, I could at any point call this and it would do the encryption. And then I would have the right value for the hash password and I could save it to the database. So for example on new.php, right here before I save it, I could just call admin and then set hash password, right? So I'll set it, and then I'm ready to save it.

And once I go to save it, it will then save that hash version because that's how we program the active record, right? All that work we did in the last chapter, it's automatically going to save that property for hash password to the database. It won't save the other one for password because it's not listed up here in our DV column, so it'll get ignored. So we could do it that way. However, I don't love this idea of having to remember to always do this. I would have to do it for creating a new one as well as for updating. I would rather if that could just happen automatically behind the scenes, and it can.

I'm going to cut that line out of there, let's go back over here, I'll close up my form fields. Let's go back over to the admin class. Now, we have in our database object class, if you remember, we wrote one here called create, right? We wrote another one here called update. What I really want is before those functions get called, to have the encryption happen automatically so that it hashes it and then it calls these methods. So we're inheriting those into our admin class.

So stop and think for a second, how can we perform an action and then have that action that's in the inherited behavior take place? We have the ability to do that. We can do it by overriding the original, create and I'll just do another one here, public function update, and I can override the behavior so that now it calls this set hash password, and then once it's done, it can call the parent version of create.

Remember this from the object oriented programming class? Where we talked about how we can call the parent object? So I'm saying do your own version of create, and then call the parent's version of create. And we also want to make sure that we have a return back after that because this is going to return a value, true or false, and we want to make sure we return that back from our overridden method as well. So this is just going to basically hash the password and then call the original version. We can do the same thing here, let's copy this and put it down here in update.

protected function create() {

$this->set\_hashed\_password();

return parent::create();

}

protected function update() {

$this->set\_hashed\_password();

return parent::update();

}

Now that we're hashing the password, let's try it out. Let's come back over here, let's create a new admin. You can feel free to use your name. I'll do mine, Kevin Skoglund. Email, I'm going to just say Kevin at nowhere dot com. Username, for me it's going to be kskoglund, and password, and I'll just put in something like secretpassword. Not the best password, but it'll get us by for now. Now create admin, and look at that. It created the admin for us. Now we don't see the password there, we're not showing it anywhere, but we can go into our consul and log into mysql.

And let's take a look. Let's say select all from admins. And you should see an encrypted password here, right? Seeing this dollar sign in front, that lets you know that this is a blowfish or bcrypted password. And then we can exit back out of mysql. Now we have more work to do still, we actually aren't logging in the users or anything like that, so we've got more work ahead of us, but we now have the ability to hash passwords when we create admins in the database.

I think we were able to do it in a pretty slick way by using this call to the parent class.

Admin validation

In this part, we'll talk about how to add validations to our admin class in an object-oriented way. You'll recall that in the last chapter, we gave our DatabaseObject class a method called validate and it just has a real skeleton of a method there, just sets errors equal to nothing, and then returns those errors. What we need to do is add custom validations to the admin class that inherits from DatabaseObject. So, in admin.class.php, we want to add a new method down here which is also going to be the same type of method, protected function called validate, and it's going to override that behavior with its own behavior.

Code looks like this:

protected function validate() {

$this->errors = [];

if(is\_blank($this->first\_name)) {

$this->errors[] = "First name cannot be blank.";

} elseif (!has\_length($this->first\_name, array('min' => 2, 'max' => 255))) {

$this->errors[] = "First name must be between 2 and 255 characters.";

}

if(is\_blank($this->last\_name)) {

$this->errors[] = "Last name cannot be blank.";

} elseif (!has\_length($this->last\_name, array('min' => 2, 'max' => 255))) {

$this->errors[] = "Last name must be between 2 and 255 characters.";

}

if(is\_blank($this->email)) {

$this->errors[] = "Email cannot be blank.";

} elseif (!has\_length($this->email, array('max' => 255))) {

$this->errors[] = "Email must be less than 255 characters.";

} elseif (!has\_valid\_email\_format($this->email)) {

$this->errors[] = "Email must be a valid format.";

}

if(is\_blank($this->username)) {

$this->errors[] = "Username cannot be blank.";

} elseif (!has\_length($this->username, array('min' => 8, 'max' => 255))) {

$this->errors[] = "Username must be between 8 and 255 characters.";

}

if($this->password\_required) {

if(is\_blank($this->password)) {

$this->errors[] = "Password cannot be blank.";

} elseif (!has\_length($this->password, array('min' => 12))) {

$this->errors[] = "Password must contain 12 or more characters";

} elseif (!preg\_match('/[A-Z]/', $this->password)) {

$this->errors[] = "Password must contain at least 1 uppercase letter";

} elseif (!preg\_match('/[a-z]/', $this->password)) {

$this->errors[] = "Password must contain at least 1 lowercase letter";

} elseif (!preg\_match('/[0-9]/', $this->password)) {

$this->errors[] = "Password must contain at least 1 number";

} elseif (!preg\_match('/[^A-Za-z0-9\s]/', $this->password)) {

$this->errors[] = "Password must contain at least 1 symbol";

}

if(is\_blank($this->confirm\_password)) {

$this->errors[] = "Confirm password cannot be blank.";

} elseif ($this->password !== $this->confirm\_password) {

$this->errors[] = "Password and confirm password must match.";

}

}

return $this->errors;

}

I'm checking the username, and then I've got some checks on the password. The password checks that we have here are going to require that the password be 12 characters long, that it have an uppercase letter, and that it have a lowercase letter, that it have one number, and one symbol. We talked about how to implement each of these validations and why we might need them in the previous course, so I won't go over them again. You can feel free to comment out or edit these if you want something that's different. And then you'll see that I've also got one here for confirm password. Confirm password is a property that needs to match password.

The reason why is because we have that form and at the bottom of our form, we have our password, and then the user has to retype the password and they must type it the same. And that way we make sure that they didn't mistype the password the first time. It's just a sanity check to make sure they don't accidentally lock themselves out with a wrong password. They have to type it twice and it has to match. Once we've got that, we can go back and let's just save this file so that we have our validations in place and now we should have validation.

If I go and just edit my admin here and I put in a password, let's just try submitting it with no password. Edit admin comes up and says, "Nope, password cannot be blank." Same thing if I take out first name. "First name cannot be blank." And let's change the password to something else. I'll just put in XYZ for now, and I'll put in ZYX down here. You'll see password must be 12 or more characters, password and confirm, password must match. Now let's try a good one here. I'm going to put in capital Password, pound sign, one, two, three, four.

Again, not a very secure password, but it allows me to quickly put something in here. Now I'll click edit admin, and it accepted it. So now it was able to meet the validations and to save the record. We also, in the essential training, talked about the fact that we don't always want to run those validations on the password. Now, if I'm adding a new admin, then of course I do. I always want to have a password, and I always want to validate it. But, when I have an existing admin, I don't always want to run it.

What if I just wanted to make a change to the user's name? What if I had misspelled the user's first name, or if I needed to change their email address to something else? Let's say it was email@somewhere.com. Well, I don't want to change their password, right? This is potentially a live user, and they have a password that they know about. I don't want to set it to something else. I don't know what their password is. So if I click edit admin, it won't let me submit it. So I need to make these validations optional when I'm updating. If there's a password there, I want to validate it.

If there's not a password being submitted, then it should assume that we're not updating the password. The way that we'll do that is I'm going to add a property up here at the top which is going to be a protected property. It's going to be called password\_required, and by default it's going to be true. A password is normally going to be required. However, when we go to update, we want to check and see whether the password has been set or not to know whether we should set that to false. So let's add a check here that says if this password is not set, meaning that it's equal to nothing, it's equal to an empty string which is the default that it would have, that would mean that we have a password and we want to validate it.

But if not, then we don't want to and so we would do that by changing set password\_required equal to false, all right? So now with password\_required equal to false, let's jump back down here to our validation and let's just add a check around all of this password validation that says if this password\_required, if it's required, then we're going to do all of these validations. I'm going to copy all of these and let's move them up and they get inserted like that.

So now we have the ability to turn those on and off. Now before we actually do it, there's one other consideration we need to make though. If the password is not being updated, then we don't want to hash the password. We don't want to take the empty string and override their password in that case so we also need to take this bit of code here and move it up here. We want to validate the password and set the hash password, right? Otherwise, we're going to skip hashing and validation, password not present, being updated we'll say.

So let's go back and let's try that out. Let's save it. We'll go back to our page and now I should be able to edit something on here like the email address from nowhere to somewhere. I should be able to click edit admin and it works. Now I'm able to update that without updating the password. However, if I go back and I edit it and I try to update the password with something, then my validations kick in and it must be 12 characters and must match the password. There's actually one validation that's missing and one more that we should add and that is that we don't currently have a validation to make sure that the username is unique.

We're checking to make sure it's not blank. We're making sure that it has the right length, but we're not validating that each username is unique. So in the next part, let's see how we can do that.

Find by username

In this part, we're going to learn how to find admin records using the username column, and then use that as a validation to confirm that a username is unique before we accept it into the database. We already have a validation function placeholder for this, go to our validation\_functions.php. There's has unique username, and that's what we had before when we were working in php with mySQL essential training. But we need to now rewrite that using object oriented programming. So, let's think about how we would do that.

The first thing we need to do, is we need to get an admin, and we would do that by asking the adman to return a user to us. So we would need to have admin class and then find the user, so we can't use Find all, and we can't use find by ID, we're going to need to write a new one, which is find by username. And we'll need to pass in the username that's been sent in to this validation, so that we can then pass that on to the class and see if we find this user by username, do we get a user back or not.

So then once we have that admin we can check and see, did it return something to us? Let's write a conditional statement here. If we got back False then we'll know that the admin didn't exist, so Admin, if it is exactly equal to False, or if we get it back and the admin ID is equal to the current ID. Right, it's possible that we're updating a record in which case it does exist in the database, we would expect it to, but we want to make sure that it's not a different record, right? We can change the user's username as long as it's not in use by someone else, and the ID is a way to check that.

So we're going to pass that in as an argument for has unique username in the case when we're doing an update. And we'll check for it here. So, if that's the case, then we'll say is unique And so we can return true. Otherwise, we'll know that it's not unique, and we will return false.

function has\_unique\_username($username, $current\_id="0") {

$admin = Admin::find\_by\_username($username);

if($admin === false || $admin->id == $current\_id) {

// is unique

return true;

} else {

// not unique

return false;

}

}

Okay, so this is our validation for has unique username, we need to write this method now, find by username. With passing in the username as an argument.

I'm going to copy all of that, going to save this, and let's go over to our admin class and let's just go up here, let's do it right below the validation. Now this is going to be a static public method, going to be called find by username, and it's going to take a username as an argument And, what is it going to do? Well it's going to find all records using the username. Now previously in database class, we found all records by their ID.

That's a good starting place for us, let's just take this code and copy it. And go over here to admin class.php, and we'll paste it. Now I'm not moving this find by username into the database object class, because it's not shared. It is unique to this class, so there's some inherited find methods, and there's some that are unique just to this class, this is one of those. So it's going to work the exact same way, all this code is the same except instead of finding it where ID equals, we're going to say where username equals, and then instead of passing in the ID, we're going to pass in the username.

Then it will execute the find by SQL, it'll check and see whether it's empty, and it'll return back a record, single record to us. So it's just going to find a single record, but either we're going to get back that record, letting us know that it exists, or it's going to be false. Alright, let's save it, and let's add this validation into our actual validations.

static public function find\_by\_username($username) {

$sql = "SELECT \* FROM " . static::$table\_name . " ";

$sql .= "WHERE username='" . self::$database->escape\_string($username) . "'";

$obj\_array = static::find\_by\_sql($sql);

if(!empty($obj\_array)) {

return array\_shift($obj\_array);

} else {

return false;

}

}

So let's fold this up, and inside the validate method we scroll down here to username, we can add a new one here, that says else if, and then parentheses not, and the name of our validation, if we come over here, is has unique username, so I'll copy that, and let's go over here and paste it in, has unique username, and we're going to pass in this username, see whether it's unique, and then as a second argument we're going to pass either this ID, if it's there, or pass in zero, if it's not there for some reason, hasn't been set, pass in zero, and that'll allow us to write an SQL statement with zero as an ID, obviously, where ID is not equal to zero, it's going to not return anything.

And then, if that's not the case, if it's not a unique username, then let's add something to the errors, I'm just going to copy this line, save me some time, and username, let's change this text to say not allowed, try another.

elseif (!has\_unique\_username($this->username, $this->id ?? 0)) {

$this->errors[] = "Username not allowed. Try another.";

}

Okay, now that we've got our validation, let's try it out. Let's save it, let's come back over here, let's try adding an admin with the same username, (unintelligible) I can just make it anything up here, bob Smith, email b at b dot com, and let's just create admin.

Username not allowed, try another. You can see my validation is now working. Now if I tried it with something else that is different, Bob Smith, create admin, Now I don't get that anymore, Right, I still get problems because I don't have a password, but I no longer get my It's not unique message anymore, so I know that my unique validation is working. What's handy about the new method that we just added here, with this find by username, is that we can reuse this same method again when a user's trying to login.

Right, that allows us to take the username on the login form and look for the user in the database by that indexed column. So it'll perform double duty, it'll work both in our validation, and when we're performing our logins.

Verify password/Login form

We learned how to create new admins in our database and how to hash their password while we were creating them. We also learned in the last part with validations how to look up a user by their username. So we're ready to start the process of building a login form. The idea is that the user would submit their username and password, will find that user by the username, and then verify that the password is correct. The way that we verify passwords are correct is with a handy peach p function called password\_verify(), and it makes it really simple because we just pass in the new password that's being tried, and match it with the hashed or encrypted password that's stored in the database. And php will handle all the work behind the scenes to see if they match or not, and it will return a boolean: true or false.

Now, it doesn't actually decrypt the hashed password during this process. What it does is it encrypts the new password using the same algorithm, and if it's the same password going through the same algorithm, then it should return the same results so that it can compare the two hashed versions and know whether or not they match. We have 2 files for login and logout that we need to create, login.php and logut.php that we will put in our /public/staff folder. Login page code looks like this:

<?php

require\_once('../../private/initialize.php');

$errors = [];

$username = '';

$password = '';

if(is\_post\_request()) {

$username = $\_POST['username'] ?? '';

$password = $\_POST['password'] ?? '';

// Validations

if(is\_blank($username)) {

$errors[] = "Username cannot be blank.";

}

if(is\_blank($password)) {

$errors[] = "Password cannot be blank.";

}

// if there were no errors, try to login

if(empty($errors)) {

$admin = Admin::find\_by\_username($username);

// test if admin found and password is correct

if($admin != false && $admin->verify\_password($password)) {

// Mark admin as logged in

redirect\_to(url\_for('/staff/index.php'));

} else {

// username not found or password does not match

$errors[] = "Log in was unsuccessful.";

}

}

}

?>

<?php $page\_title = 'Log in'; ?>

<?php include(SHARED\_PATH . '/staff\_header.php'); ?>

<div id="content">

<h1>Log in</h1>

<?php echo display\_errors($errors); ?>

<form action="login.php" method="post">

Username:<br />

<input type="text" name="username" value="<?php echo h($username); ?>" /><br />

Password:<br />

<input type="password" name="password" value="" /><br />

<input type="submit" name="submit" value="Submit" />

</form>

</div>

<?php include(SHARED\_PATH . '/staff\_footer.php'); ?>

Now let's go into the text editor and take a look at those. Let's look at login.php, and you can see that it initializes. It sets an errors array. Username and password are blank. If it's a post request, it'll try to process the login, but if it's not a post request, it'll just display the login form. You can see it submits to itself, and it just has a username and a password field and a submit to it. If it is a post request, and we've just submitted the form, then it's going to get those post values for username and password.

It's going to perform some very simple validations on them, just to make sure that they're not blank, and then we're going to try to login. If there is no errors so far, we're going to try to login. If there were errors, then none of this will take place. It'll just redisplay the form again, and the user will see those errors right here with display errors. So, if there were no errors, we want to use our new find by username method. So on admin, we call find by username, and it will either return false or it'll return an admin to us.

Now if we have an admin, we were able to get one, so admin is not equal to false, then we want to verify that the password is correct. So we need to use that verify password method. I'm going to actually put that method inside the admin class. We have set hashed password right here. Right below it, I'm going to put another one that is going to be public function, which is going to be called verify password.

I'm just going to reverse the order, pass in a password we want to try, and then it's going to call the password verify function that's built into php. It's going to pass in the password that we just sent in, as well as test it against the current hashed password. So it'll tell us whether it's true or not, and then we need to return that value back. Is it true or false? Does the password that we're trying match the one that this object has stored? Make sure that it's a public function because we're going to need to call it.

We're going to go back over to login dot php, and we're going to add after this whether or not admin is false and and check to see if the admin verify password for the password they just gave us is correct. That should return true or false, and if it is, then we want to mark the admin as being logged in. Now right now we're not doing that, so we can just test and see whether it works or not, whether or not we're able to find a user and verify the password.

Either we should see the page again if there was a problem, or we should be redirected to staff index dot php. So let's just try it. We'll go to Firefox and let's go to our new login page. So it's staff slash login dot php. There we are. I'm on the login page. It wants a username and a password. So my username, let's just try putting in some garbage first. I'll just type some random characters. Submit. It says password can't be blank, so let's type some random characters there. Submit. Login was unsuccessful, right? It wasn't able to find the username is why.

So now, I know that I have a user. Make sure you know your user's name, right? My username is kskogland. I'll hit the back arrow a couple times to get back here. Let's try kskogland with some junk password. Submit. Login was unsuccessful, but it's unsuccessful because the password was wrong this time. Let's try putting in the correct password. For me that was capital p a s s w o r d, pound sign one two three four. You may have used something different. And it worked. Now I was able to get past that page.

I didn't get errors. Instead I got redirected to the page. So we know that it worked. Now it doesn't remember that we're logged in yet. That's something we're going to have to work on in the next part.

Define a session class

In this part, we're going to define a new class called session. And we're going to use it to help us manage our PHP sessions. We're mostly going to use it so that we can keep track of whether a user is logged in in the session or not. Let's remember that the way that we're keeping track of a user being logged in, is that we're going to store their id in the session. So for example, we have a function called login(). We pass in an id, and it sets a variable in the session and then we can keep track of that variable from then on.

On all subsequent page loads, we can check that variable and see if it's been set, we know the user is logged in. So we have another function called is\_logged\_in() that will report back whether it's set or not. And then when we want the user to log out, we simply unset that variable. So that variable acts like a hand stamp, letting us know on each of the subsequent page requests after logging in, we know this user is already logged in and we don't need to ask them to log in again. It keeps track of their authenticated state.

But that's the procedural way to do it. We're going to do that with a class. So we're going to create a new class, inside the classes directory, new File, which I will call session.class.php. It's going to need to have some php tags. And we know that it's going to also be a class called Session. And the most important part of this class is that it's going to have that property for the id. So I'm going to make a private property called admin\_id.

It can't be accessed from outside the class, it's private, but we can access it inside the class using a couple of different methods. We know we're going to need one method that's public function login. We'll pass in an admin object. We don't have to just pass in an id, we'll pass in the whole admin object. And then we'll make another public function that's going to be called is\_logged\_in(), that'll test whether or not the user is currently logged in. And one more called logout().

And that will logout any currently logged in user. So the way that we login a user is we check to see if we've been given a user, first of all, just as a good safety check. And if we have, then we want to call session, admin\_id equals whatever the admin id we were given was. Now we don't want to just store it there, though. We also want to make a copy of this line, because we have a property now, and we want it to also exist in that property. So we don't have to go back to the session. It's stored in the session, but it's also stored in our property of the session class. If we have an instance of the session class, it will have that. And then I'm going to have it return true. One security feature is here we want to always call session\_regenerate\_id(), that's going to protect against session fixation attacks. There we go, prevent session fixation attacks with session\_regenerate\_id().So it'll regenerate the session every time that someone logs in again. Okay, so once we've got a user logged in, we've stored the fact that they're logged in in the session, and we're keeping track of it in the property, as well.

public function login($admin) {

if($admin) {

// prevent session fixation attacks

session\_regenerate\_id();

$this->admin\_id = $\_SESSION['admin\_id'] = $admin->id;

$this->username = $\_SESSION['username'] = $admin->username;

$this->last\_login = $\_SESSION['last\_login'] = time();

}

return true;

}

Now we can check to see is the user logged in. Well, is the user logged in? We just check and see, is that property set? Isset this admin\_id. Right, we don't even have to look at the session file. We just look at the object's property to see whether or not it's been set.

public function is\_logged\_in() {

return isset($this->admin\_id);

}

And then for logout, we want to logout someone, we need to unset those values. So we use unset, and I'll just copy this up here so I don't have to retype it. Then we'll unset that. Then let's unset this admin\_id, as well. So we'll unset it both in the session, where we're keeping track of it, and also in the property. And then let's just return true.

public function logout() {

unset($\_SESSION['admin\_id']);

unset($this->admin\_id);

return true;

}

Okay, so that will allow us to log in. We'll be able to tell if someone is logged in, and they're able to log out.

But on subsequent page loads, we need to also take a look at the session to see whether they were logged in previously. So let's make another function. This one's going to be private. And this one's going to be called check\_stored\_login(). And what it's going to do is it's going to check and see if that value is set in the session. So if isset, and I'll paste it in there, I still have it on my clipboard. And then if it is set, then what do I want to do? Well, I want to set this admin\_id, the property, equal to it.

Right, there's a new page. I have a new object. This is nothing set in the past, right, I haven't just recently hit the form and called login. So I need to set that value there, and then I need to call this automatically, whenever a new page loads.

private function check\_stored\_login() {

if(isset($\_SESSION['admin\_id'])) {

$this->admin\_id = $\_SESSION['admin\_id'];

}

}

Now I could put that in initialize, but I think a better thing to do is to add a public constructor here, public function construct. And whenever I create a new session object, a new instance, let's automatically perform this.

Let's automatically do this, check\_stored\_login. So I have to go down here. Copy it, this check\_stored\_login function happens automatically.

Now if we're going to use the session, we need to also have a session. So we need to make sure that sessions are enabled. And in initialize.php, we already have a line here, session\_start, it will do that, but it's commented out. I'm going to actually copy that part and take the line out of the initialize and put it in my session class. I'm even going to put it inside the constructor.

public function \_\_construct() {

session\_start();

$this->check\_stored\_login();

}

So as soon as I call a new session object, and try and create a new object, it'll start the sessions for me, and then immediately go check and see if the user is logged in or not. Let's save all that. Now I just need to create a new session object. So let's go back to initialize.php. And down here at the bottom, after we have our DatabaseObject, let's create a new Session Object. Session equals new Session.

$session = new Session;

That's all we have to do. Now I have a session object that I can talk to, I can tell it to login, to logout, find out whether a user is logged in. Once we have that, we can go to login.php, and when we want to mark the user is logged in, what do we do? Well, we tell the session instance to call its login method, and pass in the admin object.

That's all there is to it. We just tell the session, hey, login this user. I'm going to copy that and then save the file, because I also want to go over to logout, and let's do the same thing. But instead of calling login on the admin, we're just going to call logout, with no arguments at all. Does the exact same thing, so that will call the logout method on the session. So now we're ready to try it out. But we don't have any good way to indicate whether we're logged in or logged out, yet. So let's add something here. Let's go to the shared/staff\_header page, right, so inside private, inside shared, the staff\_header.

And on that page where we have some navigation, let's actually put a conditional statement around this li tag. So let's do php tags if the session is\_logged\_in, then we're going to perform one set of actions, otherwise, php we're going to perform another one. So that'll show menu and then let's also just duplicate this line, and add this as a link to logout.

<ul>

<?php if($session->is\_logged\_in()) { ?>

<li><a href="<?php echo url\_for('/staff/index.php'); ?>">Menu</a></li>

<li><a href="<?php echo url\_for('/staff/logout.php'); ?>">Logout</a></li>

<?php } ?>

</ul>

Logout, there we go. Now we should see these two links when we're logged in and we won't see them when we're logged out. So let's save that. So let's go back to our menu page, just to start with, and you'll see that we don't have those items up here. All right, so now let's go back to the login page, login.php, and let's try it. Let’s login with our username and password.

Click Submit. And now I'm logged in. Menu and Logout. If I click logout, now I'm logged out again. So those give us a sort of temporary way to tell whether or not we're logged in or not. Eventually what we want to do is we want to have page access controls where we're actually not able to go to some pages if we're not logged in.

Add page access control

Now that we have the ability for users to log in and log out and we're able to remember in our session whether or not they've logged in or not, we're able to now add access control to our pages. That is, that we can restrict the pages in our staff area to only allow logged in users to see them.

I just go and create new function require\_login() and put it in one of our functions. Your status\_error\_functions.php, let's put it in here. Right here at the top, I'm going to make a new function which I'm going to call require\_login() and inside there I'll just put in this code.

function require\_login() {

global $session;

if(!$session->is\_logged\_in()) {

redirect\_to(url\_for('/staff/login.php'));

} else {

// Do nothing, let the rest of the page proceed

}

}

So now with this function, all I have to do is call this require\_login() function and it should do it. I'll save it. Let's go back over to index.php and I'll just put in require\_login.

Let's now take that same bit require\_login. I'll just copy that line and we don't need it on our login or logout pages, but all of our bicycle pages should get it. So let's open up delete and right after we initialize that's where we want to do it. I don't need php tags here because I'm still inside php tags. And let's see, edit, same thing, I don't need the php tags. Those will just break it so let's remove those.

And form fields, won't need it. Index.php, I can put it right here. On new.php, put it right after initialize and remove those php tags. And I'll do it for show right here. And then very quickly let's go to admins, I can do this for show and index, those are the ones I think that had the ability to just paste it. On the other ones delete, have to remove the php tags.

So let me do that this time and then I'll copy this line as is and we'll go to edit, paste it in, and new, paste it in. Okay, so now I have those access controls on all those pages. I'm going to just close up a whole bunch of pages here. Let's quickly close those back up. Now let's go back over and let's try it. So if I go to staff/admins now index.php, I shouldn't be able to get this page. Nope, sent me back to the log in page and that should be true for all of my pages. But if I log in, kskoglund with Password#1234, submit it, now I am able to go around and browse the different pages.

Now it works as expected because I'm logged in. It remembers that I was logged in. We've successfully implemented user authentication using our object-oriented programming techniques. And now we have a password-protected staff area for managing our admins and our bicycles.

Improve the session class

Before we leave this topic completely, I want us to make a few improvements to our session class. Hopefully this will help to illustrate why it's useful to wrap up some of that session functionality into a class in the first place. Putting all of that code into the session class allowed us to have it all in one place, but it also allows us to be able to extend this class and to do more work on it in the future, so it would become more complex over time and helps us to manage that complexity. For example, let's say that we wanted to have the current user's username displayed up here at the top, so it would just say who you're logged in as.

Well, we could add that feature by simply adding a new public attribute for username. Let's say that we also wanted to keep track of when the user last logged in. We'll make a private attribute for their last\_login. The idea might be that we don't want to allow users to stay logged in forever. At a certain point, we want their login to expire. So let's add those two features in. When a user first logs in, we need to keep track of their username, so we're going to set this username equal to the admin username.

And we know that we also want to keep track of the last\_login which is going to be equal to the current time. Now, we don't want to just store those in the properties. We also want to store them in the session just like we did here. So I'm actually going to take this and we can actually do a double assignment here where we just assign one and then right away assign the other one. This one's going to be username and this one's going to be last\_login, so it'll assign it both to the session and to the property in sequence. We can actually do the same thing here if we wanted just to keep these parallel.

I'll put that up there and now those all look the same. Okay, so that's when you first log in. It's going to keep track of them. When you log out, we also know that we want to get rid of them, so we want to unset those values so I'll copy this line and just paste it in, username and last\_login. And same thing for this, we want to remove the property values as well, so this is username and last\_login.

And then when we go to check a stored login to see if a user has previously logged in, we also need to get those values at that point as well, so let's put those in. It's also going to be username username and last\_login last\_login. Okay, so now as soon as we log in, we should be keeping track of that additional information, and that means that then we have the ability to use it. So let's trying using it. Let's go to our staff\_header.php and right after where we see if the user's logged in or not, let's just output their username.

<ul>

<?php if($session->is\_logged\_in()) { ?>

<li>User: <?php echo $session->username; ?></li>

<li><a href="<?php echo url\_for('/staff/index.php'); ?>">Menu</a></li>

<li><a href="<?php echo url\_for('/staff/logout.php'); ?>">Logout</a></li>

<?php } ?>

</ul>

I'll make a new line here, a new li, list item, and then we'll do user colon and then inside php tags, we will echo back the session username. Now, we should have that value because the user is logged in at that point. So let's just save that. Let's go try that out. Login with our username and password.

So now on each and every page, it remembers what my username is. It doesn't have to go to the database and look it up. It's stored in the session and it's managed by my session class. It makes it nice and easy. What about that last\_login? Let's say that we wanted to check and see whether the last\_login was recent or not. Our session class can be given that ability. We can add another method here. We'll make it a private function called last\_login\_is\_recent.

And then inside here, let's check and see what the last\_login was. So if there is not a login, if it's not set, well then we know that it's obviously not recent. That's just a good error checking thing. So if there's no last\_login, then let's return false, elseif and we'll do a test here to see whether it's over the maximum age. But before I do, I'm just going to finish my thought here and just say that the last one, this is going to return false if it's too old.

Otherwise, the last thing is return true. It passed both of our tests. It was present and it was not too old. So how do we check and see if it's too old? Well, we check that value of last\_login and we need to add a value to it. So the value I'm going to say up here is a constant. I'm going to put it in here. We'll make it a public const. We'll call it MAX\_LOGIN\_AGE and then we can set this to a value, 60 times 60 times 24, that'll be one day.

If you want, you could multiply it by more if you wanted to have the number of seconds in one week or three days. You can choose what that would be. So I'm going to say one day is the maximum login age. I'll copy that constant that I've defined. It's a class constant so I'm going to say the last\_login plus self colon colon and then the MAX\_LOGIN\_AGE and then I can check to see whether that value is less than the current time. Let's just put another set of parenthesis around this addition.

There we go, so we'll take the last\_login time. We'll add the number of seconds for the maximum login. And if it's less than the current time, we'll know we've already passed that time, right? We'll know that we've already gone past it. We have already achieved more seconds than that represents and so it's false. So then maybe you have last\_login as recent, then add it up here to checking whether the user is logged in or not. And let's just copy this line. I'm going to comment this one out so we still have it if you want to go back to it. But just to show you what that might look like, we could say well if the user is set and the this, and what's the name of our method down here, last\_login\_is\_recent, so if the user is set and last\_login\_is\_recent, then the user returns true.

So if we save this, let's go back over here, let's see if our page still loads, click on the menu, it still works. I'm on my menu page. I'm able to move around because my login is still recent enough. If we change this so that the login time was much less, let's just temporarily make it one second, now I come back and when I go and reload the page, it says nope, you got to log in again.

Code for session class now looks like this:

class Session {

private $admin\_id;

public $username;

private $last\_login;

public const MAX\_LOGIN\_AGE = 60\*60\*24; // 1 day

public function \_\_construct() {

session\_start();

$this->check\_stored\_login();

}

public function login($admin) {

if($admin) {

// prevent session fixation attacks

session\_regenerate\_id();

$this->admin\_id = $\_SESSION['admin\_id'] = $admin->id;

$this->username = $\_SESSION['username'] = $admin->username;

$this->last\_login = $\_SESSION['last\_login'] = time();

}

return true;

}

public function is\_logged\_in() {

// return isset($this->admin\_id);

return isset($this->admin\_id) && $this->last\_login\_is\_recent();

}

public function logout() {

unset($\_SESSION['admin\_id']);

unset($\_SESSION['username']);

unset($\_SESSION['last\_login']);

unset($this->admin\_id);

unset($this->username);

unset($this->last\_login);

return true;

}

private function check\_stored\_login() {

if(isset($\_SESSION['admin\_id'])) {

$this->admin\_id = $\_SESSION['admin\_id'];

$this->username = $\_SESSION['username'];

$this->last\_login = $\_SESSION['last\_login'];

}

}

private function last\_login\_is\_recent() {

if(!isset($this->last\_login)) {

return false;

} elseif(($this->last\_login + self::MAX\_LOGIN\_AGE) < time()) {

return false;

} else {

return true;

}

}

}

I'll change this back to say it's long enough. We'll reload index.php and we're able to get back there again, so we have the ability to control the maximum login age. The fundamental idea I'm getting at is that we can add more complexity over time to this class and it makes sense where it goes. It's easy to manage this complexity. For example in the future, you might want to check and see when a user logs in what their IP address is or what the user agent string is that they're using and you might want to use those in some way as well.

And the session class becomes the perfect place for you to do that.

Session message

In this part, we'll look at how we can improve our session class by allowing it to manage our session messages. When I talk about session messages, what I'm referring to is the fact that in some of our pages like edit.php, once we have successfully updated one of these records, we have session message equals the admin was updated successfully. And then we want to display that on future pages as well. So the way that we do that is with the staff header, we need to have something here that says display\_session\_messages and that will tell us those reports of whether or not something has been added to the session.

We have this function already existing, display\_session\_message() exists inside our status\_error\_functions.php . And if we scroll down here, you'll see what it does. It calls get\_and\_clear\_session\_message. If it's set, then it displays some HTML with the message inside of it. Get and clear it both finds out if it's there, retrieves it, and unsets it from the session. Now that all works just fine and we could leave it as is. But now that we have this session class, that seems like a better place to put these messages so let's add them there.

I'm going to go into the session class and I'm going to make a new method down here which is going to be a public function and I'm going to call it message and it's going to take an argument. And the thing is I want to be able to call message either by passing in a string to set a message or if I call this method without any value sent in, it'll default to the empty string and I will read back the message instead of setting it.

It's going to serve a dual purpose. So if it's empty message, msg, then this is a set message. It's not empty and else will know then this is a get message. Okay, so if it's a set message, then what do we want it to do? Well, we just tell it that the session message is going to be equal to whatever message was passed in and we can just return back true at the end.

Now if it's a get message, then we want to get the value that was set there so I'll just copy this and we'll just return whatever the session had. So that allows us just to use one method to both set and retrieve the messages.

public function message($msg="") {

if(!empty($msg)) {

// Then this is a "set" message

$\_SESSION['message'] = $msg;

return true;

} else {

// Then this is a "get" message

return $\_SESSION['message'] ?? '';

}

}

public function clear\_message() {

unset($\_SESSION['message']);

}

So we can go for example to our admins edit.php page and here where before we had session message equals, we can instead change this to be session object to set its message equal to this.

That's the argument we pass in. Let's take that, I'll copy it. Let's go to new. We do the same thing there. I'll paste it and then the admin was created successfully. That'll make it look the same. And I think we did it with delete. Yup, was deleted successfully. And then I've got the same thing on bicycles. I'll paste it in here, the bicycle was deleted successfully.

I'll copy that one so that I can use that again on edit.php, it was updated successfully. It was created successfully. Okay, so that takes care of setting the message. In each of those cases, I'm now using my new method to do that. Let's save all those. Let's make sure I save them as I go. Some of those weren't saved as I went.

Okay, so now what about this get\_and\_clear\_session\_message business that I was doing before? I can rearrange this a bit now because I have that method there. Instead of getting and clearing the session message here, let's just get the message by calling dollar sign session and then message. Now remember, I don't have access to this variable inside this function unless I call global dollar sign session. That'll bring it in so that I have access to it. Now I have my message and then I can use the message so if the message is set and it's not equal, then we'll display it.

We also need to clear it, right? We also need to make sure that we're clearing it. We were doing that before inside get\_and\_clear\_session\_message. I'm going to take this line right here and just copy it. I'll go back to session class and let's write another method here. It's going to be called public function clear\_message and all it's going to do is just call that unset on the message to make sure that it's clear. All right, so now I can go back to my error functions and I can just tell it to clear the message.

So if we get it and it's not empty, then clear message. There we go. That should work.

function display\_session\_message() {

global $session;

$msg = $session->message();

if(isset($msg) && $msg != '') {

$session->clear\_message();

return '<div id="message">' . h($msg) . '</div>';

}

}

I can now remove this function completely and it should just work as is because I'm already making calls to this existing function. So let's go back here, let's reload our page to make sure this is still working.

All right, so let's try editing something. Let's go to our bicycles. Let's take this last bike here and let's just change the price to be 199 instead. Edit bicycle, there we are, and the bicycle was updated successfully. It worked. Click back and now the message is gone. It's no longer in the session so it only displays one time and then it disappears after that.

My hope is that you see how objects allow you to group similarly-themed code together into one area and also how it allows you to manage complexity as that code continues to grow.

Object-Oriented Pagination

Pagination concepts

In this chapter we're going to learn how to do object-oriented pagination. We will begin by talking about some of the underlying concepts that make pagination work. The reason we need pagination is to manage large sets of data. Let's imagine that instead of having just 10 bicycles in our database, we had 1,000, or even 10,000, or 100,000. We probably would not want to view all of those bicycles on a single page at one time. It would be visually overwhelming, but it also requires server to do a lot of work to retrieve that many records from the database, and then to instantiate them, and turn them into PHP objects.

It would take a lot of time and a lot of memory. We probably don't want to do that. So instead, pagination allows us to just show a subset of records, and then, when we want to see more records, there are links that will take us to other pages of the results. Typically you would have previous or next to take you to the previous or next pages of results, or you might have a list of numbers showing you the different pages so that you could jump directly to page seven. It's also common to display some information to the user about how the complete set compares to the subset that they're seeing.

You might tell them how many records there are, how many total pages there are, or how many records they're viewing per page. You've certainly come in contact with pagination before, if you've ever searched for anything online. If we search for the word pagination on Google, and you can see it tells me there are 71 million results. Now I certainly don't want to see all 71 million results in one long scrolling page. Instead I'm just going to see a subset of those records. We scroll down, and once we get to the bottom of the current page, you can see that we have pagination links down here at the bottom.

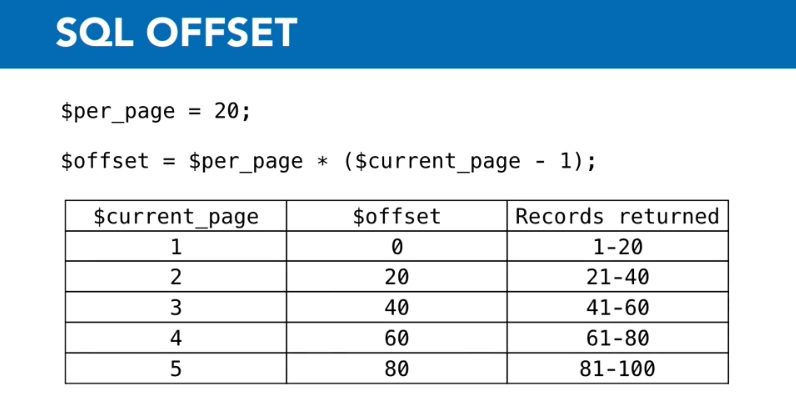
If I click Next, it takes me to the next page. Then it tells me that I'm on page two of those records. If I scroll down again, you'll see that now I have a Previous link, and I also have numbered links that'll let me jump directly to a page further down the line, and it indicates my current page here by making one of the o's in Google a red color. This is the basic idea, and it's a common way to handle pagination on the web. In order to code pagination, we're going to need to keep track of three variables. We're going to need to know the current page, the number of records per page, and the total record count.

If we have those three pieces of information, we'll be able to calculate everything else that we need to know for pagination. For example, the previous and next pages would be the current page plus or minus one. The total number of pages is going to be the total count of records divided by the records per page, and knowing the total number of pages will let us create links to numbered pages, and tell us if we've reached the last page or not. We're also going to need to make use of a couple of features of SQL.

We're going to need to use SQL's limit in order to set the number of maximum records that should be returned by the query. That's essentially the number of records we want per page. We just want SQL to return a page of results. We don't want everything, just what we're going to be putting on the current page. And then we're also going to use SQL's offset in order to skip over records that we don't want. That allows us to see pages beyond the first page. There's a formula for figuring out the offset. It's the number of records per page, times the current page, minus one.

Let me demonstrate how that works. Let's say that we have per page equal to 20. That would mean that then the formula for offset is per page, which is 20, times whatever our current page is, minus one. So if our current page is one, we plug it into that formula. One minus one is zero. 20 times zero is zero, so the offset becomes zero, so the records being returned would just be one through 20, a page of results. When we go to the second page, though, now current page is two, so it's two minus one, which is one, times 20 for the per page, gives us an offset of 20.

So now we're going to skip over the first 20 records. We're going to skip those. We've already seen those on the first page, so we're going to tell SQL ignore the first 20 records, and start giving me records at 21, so we'll see 21 to 40. If current page is three, the offset becomes 40, and the records returned are 41 to 60, and so on. And this makes sense when you look at the fact that at the end, on page five, we're at records 81 to 100. We have 100 records, 20 records per page, spread across five different pages.



Limit and offset are what allow this to happen. So the SQL is going to look something like this. To retrieve a page of results, select all from bicycles, and then we provide the limit, and we provide the offset. We're also going to need to use one other part of SQL, which is in order to get a record count, we're not going to get back all of the records. We're going to use count, so select, and then you write the word count with parentheses around the asterisk, and then we don't get back records, we tell SQL to count them for us, and we just get back a row that contains the count.

So those are the underlying concepts. Now let's start writing the code to make this happen in an object-oriented way.

Define a pagination class

In this part, we will define a new class that can help us to keep track of our pagination variables and calculations. So let's create a new class. This is going to go inside our classes directory. Make a new file, and I'm going to call it pagination.class.php. It will need some php tags, and it's going to be a class definition for pagination. So inside this class, we want to keep track of the different properties. And we know that there are three main variables we want to keep track of.

Public current page, public per page, and public total count. So the way that I think that we'll add those properties in here is when we create a new instance of the pagination, we can just pass those in while we're creating it and then it will set them. So let's define a construct method that'll do that for us. Public function\_\_construct and then we'll pass in the page=1 and per page is going to default to 20 and $total\_count=0.

Okay, so then we have this current page, should get set to the value of page, and I'm also going to make sure that this gets turned into an integer. I'm just going to type cast that by putting int in front of it because if this is coming in from the URL string which is where we're going to be finding on our page later, it's going to be a string.

This'll just make sure it turns it into a number so we can do math with it. And then I'm going to just paste that two more times because we're also going to have per page and total count and those values are going to be per page. Oops, I lost the dollar sign, and total count. There we go. So that's going to set all of the different properties for us. Now lets think about some of the other methods are that we might want here. We know we have a calculation for figuring out the offset, and it's very handy to put here public function offset.

Do you want to know what the offset is? Well you just ask it and it will do that formula for you. So going to return back this and it's per page times then in parenthesis this current page minus one. That's the formula for offset, so that'll figure that out for us. We give it the current page and the per page and we'll know what the offset is. We also can figure out the total number of pages. Function total pages, and we know how to figure that out it's going to be return back dollar sign this total count divided by this per page.

Now there's the possibility that it won't divide evenly in which case we want to round up. So we can use C-E-I-L for the ceiling. That'll round up, always. So it'll always round up to that last page so if we have a partial page it'll make sure that it get included as being the last page. And then we have previous page and next page. Lets do next page first. Public function next page The next page we know is just going to be this current page plus one, right.

But let’s not just return that value, instead let’s keep track of it for a second. Next equals, and then let’s check to see, is that value greater than the total number of pages that we have? So we can do that by saying return back if next is less than or equal to this total pages, and if it is then we're going to return back the value, right. If it's less than the total pages we'll return the value otherwise return back false.

It's just going to be false, so there is no next page. Right, so it's either going to return a number from one up to the total number of pages or it's going to return false. And we can copy that to do for previous as well and I'll just put it right here. Previous page, it's going to be a similar idea current page minus one, and then they'll catch it in a variable called prev and then we're going to test though to see whether prev is greater than zero. If it's greater than zero then we have a previous page.

Right, if it's zero or less than zero it's going to be false there is no previous page. So that's it. Now we have a class that'll do some of this work for us. Lets save that and lets open up a page that we can do some pagination on. Let’s do it on the staff area page for the bicycles. We'll do index.php and what we're going to do is we're going to put our pagination right here at the top. We're going to paginate these. Right now we're finding all of them. We're not going to do that.

class Pagination {

public $current\_page;

public $per\_page;

public $total\_count;

public function \_\_construct($page=1, $per\_page=20, $total\_count=0) {

$this->current\_page = (int) $page;

$this->per\_page = (int) $per\_page;

$this->total\_count = (int) $total\_count;

}

public function offset() {

return $this->per\_page \* ($this->current\_page - 1);

}

public function total\_pages() {

return ceil($this->total\_count / $this->per\_page);

}

public function previous\_page() {

$prev = $this->current\_page - 1;

return ($prev > 0) ? $prev : false;

}

public function next\_page() {

$next = $this->current\_page + 1;

return ($next <= $this->total\_pages()) ? $next : false;

}

}

Instead we're going to use pagination instead. So we know that we're going to want to create one of our pagination objects. Pagination equals new pagination, and we know that we want to pass in those values to it. We want to pass in the current page, and we want to pass in per page, and we want to pass in total count. So we need to be able to find those values. We'll work on adding that in the next part.

Paginate bicycles

Now that we've defined our pagination class, we're ready to use that class to help us to paginate our bicycles. So where we left off in the last part, was we had added this pagination class in here, but it's not going to work yet and the reason why is because we're parsing in arguments that don't exist, current page, per page and total count. We need to find those values. I'm just going to copy those and let's go up here and for each one of these, we know we need to figure out what the value's going to be equal to. Current page, per page and total count. So current page, we're going to get that from the URL.

The idea is that when we click on one of the links that will eventually take us to the next page or jump to a page, it's going to be the same URL as what's up here now, but it'll add a query parameter to it that looks like this. Page equals two. What we want to do is be able to grab this value from the URL and use that as the page. So let's go back over here. We know how to do that using the get super global to get page, and if there's not a page, then the page is one, we're on the first page.

So let's default to one using the double question marks. So if there isn't a value for any reason, the page will just be one. So that'll be our current page. Per page can be any value we like, it's up to us, we decide how many items we want to show per page. Normally you might pick a big number like 10, 20, 30, something like that, but because we have seven bicycles here and we want to see pagination, we want to pick a number that's less than seven, so that we get multiple pages of results. So I'm going to pick five.

So we'll just have two pages of results, that'll be five per page; the first page should have five results on it, the second page should have two results on it. So that gives me my per page number. Now what about total count? How can we get a total count on the number of bicycles? We could use something like bicycle find all, but the problem with that is that goes and finds all the records, and there might be 10,000 of them. And it's also going to instantiate all of those records, which would return an array with 10,000 objects in it, and that's not what we want.

Instead, we want to write that more efficient sql statement that will count the records for us. I could just write the sql statement here one time, but it's something that I may use more than once, so I'm actually going to go ahead and add it to my database object class that gets inherited by all of my other classes, like bicycle and ad map. So let's go to the database object class and open that up, and then right here below find all, I'm going to copy that, paste it in here, and this is going to be called count all.

static public function count\_all() {

$sql = "SELECT COUNT(\*) FROM " . static::$table\_name;

$result\_set = self::$database->query($sql);

$row = $result\_set->fetch\_array();

return array\_shift($row);

}

So we're going to count all of the records and all of sql's going to be pretty much the same, except instead of finding all records, we want it to do a count instead. Putting count and then parentheses around the asterisk will tell sql, just count the rows, I don't want the rows back, just count them. So I'm going to get back a single row that has a single column with a single value in it for the count. So I don't want to call find by sql, because that's expecting to get rows of data that I can use to instantiate objects.

Instead, I want to change this and just tell the database connection that it should execute a query. So I'll do that with self::database and then let's tell it to do a query using that sql. And I don't want to just return it, because again, that's a result set that I'm getting back. So we'll call result\_set and then I want to take that result\_set and I want to get the first row from it. You know how to do that. Normally with fetch assoc, but instead here, I'm going to be doing it with fetch array.

I don't need an associative array, it's just a single value, there's no reason to have a label on it, so I'll just use fetch array to get back an array as a row and then I need to get the first value out of that array. That's the actual count. The number that's in there is in that first row. So it's going to be return and we'll use array shift to grab that first value out of the row. It's a little bit of a convoluted way to get something that's so simple like a count out of it, but that's just the way sql returns it. It returns a result set with one row, that one row has one column and one value.

So we have to go through these steps to get to that value. Now that we have count all defined here, we can jump back over to index.php and we can use it. Bicycle::count all and that will return the count to us. Now we have the three variables we need for our pagination object, now we have our pagination. The next thing, of course, is to limit the results based on those calculations. Now, we could do this a couple ways. We could modify find all so that it took some parameters, and it would allow us to parse in a limit and an offset.

Instead of doing that, I'm just going to write some new sql here, that's going to select all from bicycles and I'll append it with limit space, and then we're going to drop in interpolate the per page value that we know. Put a space at the end and then sql equals. And now let's do offset, and now we need to provide the offset.

Well, guess what knows our offset? Our pagination class. Pagination offset. It's got to have parentheses after it, because we're calling a method. And then we have our sql. So now we just need to use that sql. We can do that with bicycle::find\_by\_sql, which is the same thing that find all users behind the scenes. We can just run custom sql with it and it will take that sql, get back rows of data and instantiate those and turn them into bicycles. So I get the same result.

I end up with an array of bicycle objects just like I did with find all. But now, I'm getting a paginated version. Let's save it and let's try it out. Let's go back over here to this page, and before we try page two, let's just try page one. So I'm going to erase this. We'll just go to bicycles index.php and look at that, I get five. That's exactly what we'd expect. That's the first page of results. Now there are more results, they're just on another page. So if I do ?page=2, now I get the second page of results, six and seven.

$current\_page = $\_GET['page'] ?? 1;

$per\_page = 5;

$total\_count = Bicycle::count\_all();

$pagination = new Pagination($current\_page, $per\_page, $total\_count);

$sql = "SELECT \* FROM bicycles ";

$sql .= "LIMIT {$per\_page} ";

$sql .= "OFFSET {$pagination->offset()}";

$bicycles = Bicycle::find\_by\_sql($sql);

Now what I don't have at the moment are links to make this process easy. Instead, I'm having to go up here and type out the URL. It's much better if we have links that allow users to navigate from page to page. We'll create those in the next page.

Previous and next page links

Now that we have a working pagination class and we're able to paginate the records we get back, we need to add some links so the user can navigate from one page to another. And we're going to start by adding previous and next links. So far we have pagination, but to get the pagination to work, we have to actually change the URL from page one to page two and from page two back to page one. What we want are links that send this same URL which are here on the page for the user to click on. So right below the Bicycles table, we're going to have previous and next links.

Let's go into our code on index.php and let's go down below the table, and let's add some php tags, and let's start trying to write the code that we want to go here. Now we want to have this pagination, but we don't always want to have the pagination. For example, what if we had a page that only had four bicycles on it, right? In that case, we don't need pagination. We don't have more than one page. So let's wrap our code in a conditional statement. If the pagination total pages is greater than one, then we want to execute our code.

if($pagination->total\_pages() > 1) {

echo "<div class=\"pagination\">";

$url = url\_for('/staff/bicycles/index.php');

echo $pagination->previous\_link($url);

echo $pagination->next\_link($url);

echo "</div>";

So if we only have one page, no reason to show any links to navigate to a different page. Next, let's wrap our pagination in a div. So we'll have a nice way to address it in our CSS. Div class equals. I'm going to do backslash in front of double quotes like that. And the class I'm going to use is going to be called pagination and let's go ahead before we forget and create our ending div tag. Okay, so that will give us some div tags and then inside there we want to provide the links.

So let's do our next page link first. So we know what that's going to look like. It's echo, it's going to be a link, so a href equals backslash backslash and then it'll say next, right, that's the basic idea of it. I'm also going to spruce it up a little bit by after the next I'm going to put arrows as well. Now you could just do greater than greater than, but I'm actually going to use an HTML entity here which is ampersand raquo and then a semicolon. That's going to give us a nice slightly smaller version of two arrows pointing to the right.

Okay, so this is what our link is going to look like. I'm going to go ahead and break it right here. And just go to a new line, like this. Just so I have a little more room here for constructing this URL. Where do we want this to go to? We know from the pages we were just looking at in the browser that we want it to say page equals two when we're going to page two. So we could just put that here, and it would by default use the current page, but I'm actually going to specify the URL. I'm going to put a URL here and I'm going to actually start by creating the URL up here on this line.

So it's going to be URL four. I use my URL four helper function and then staff bicycles index.php. Okay, so that's the page I want to go to. So let's put that here. I can use the curly braces and then I can interpolate that string. And then for the page, it's not always going to be page two. We want that to be dynamic. So again let's use some curly braces and inside there let's put whatever the next page is. We know what the next page is. We have a method in our class here called next page.

It either returns the current page plus one or if it's greater than the number of total pages, it's going to be false. So it's going to be either the next page or false. So let's use that. Let's use pagination next page. Now there is a possibility that it will be false which means we're on the last page. There's not a next page. In which case we don't want a next page link to appear at all. So let's copy this and let's wrap the whole thing in a conditional statement that if next page is not equal to false, then we want to display the link.

But if it's equal to false, well there is no next page, and there's no reason to display the link. Let's try all this out. See if it's working. Save our page. Let's go back over here. We're on page one. So let's reload our page. And there's our next link. When I click it, it takes me to page two. Now I need a link that will take me back to page one. So let's copy everything that we just did. And let's change this to previous page and previous page and the text is going to say previous and we want the arrows to point in the opposite direction.

So I'm going to just cut that, put it at the beginning, and I'm also going to change it so that it's left-angled quotes instead of right-angled quotes. Alright let's go back and let's try it again. Reload our page and there's our previous link. Click on it, we're back on next. Click on that, we're back on previous. If we had more than two pages, then we potentially would have both of these at the same time. If we were on page two, we would have a previous link back to page one and a next link that would take us to page three. We now have a way to navigate.

Before we move on though, I want us to consider if we want to be able to reuse this often, it probably makes sense to put it someplace that we can reuse like a function. And since it's related to pagination, I'm actually going to take it and create it as a method on our pagination class. Let me show you how we can do that. I'm going to take the code for both our previous and our next page, and I'm just going to cut those out of there. Let's go to our pagination class and I'm going to make public function and we're going to call this previous link.

And I'm going to paste all the code in there for now, but I'm also going to make another method down here called public function next link. And I'm going to move this code down there. There we go. So now I've got a previous link and a next link. So the idea is that I should just be able to call the pagination's previous link and it will return everything I need for this previous link.

We need to make a few modifications though. We know that it's going to need the URL. It doesn't know that automatically so I need to pass that in. I'm going to give it a default value of an empty string, URL equals. That will give me the URL. And instead of using pagination now, this is going to use this. Dollar sign this, dollar sign this, and dollar sign this because I'm inside the instance. And I also think that instead of echoing from a class, it's a better practice to return a value and then echo it from there.

So I'm also going to change these into make it link equals and we'll make this one link.equals and then at the end we need to return back that value, return link. I'll copy that line. Paste it down here. And I'll also just copy this, save myself some typing. There we go. So I think that's a little better practice as well. Okay, so now let's use those. Let's go back over here to our index page and we should just be able to say tell the pagination to output its previous link and pass in the URL.

And we'll need to echo that because we weren't doing that before inside the method so we'll do it outside and then we'll change this to be next link, alright. Let's save it. Let's take a look at it. Reload it, oh, we have a problem, undefined variable link in pagination class line 46. Where is my mistake here? If previous page is false, so I need to have link equal to nothing in case there is the possibility it would be nothing and then let's append to both of that.

public function previous\_link($url="") {

$link = "";

if($this->previous\_page() != false) {

$link .= "<a href=\"{$url}?page={$this->previous\_page()}\">";

$link .= "&laquo; Previous</a>";

}

return $link;

}

public function next\_link($url="") {

$link = "";

if($this->next\_page() != false) {

$link .= "<a href=\"{$url}?page={$this->next\_page()}\">";

$link .= "Next &raquo;</a>";

}

return $link;

}

So now I have a value for link even if the condition is not met. There we go. Alright let's reload our page. And it works. We have previous and we have next. Go back to previous and we have next. And now it's in a reusable method in a place that makes sense, inside our pagination class.

Numbered page links

In addition to having previous and next page links, it's very common if we have a lot of results to have numbered pages as well. So we can jump directly to the page that we want instead of having to page through each one of the pages in sequence to get to the results we want. In this movie, we're going to learn how to create those links. I'm going to put my numbered links in between my previous and next link just like we saw with Google, right? Google has all the Os in their name and they have previous and next links on either side. That's the idea here. So we want to make a loop.

So what is our loop going to be? We want it to be from one all the way up to the total number of pages. So let's write a loop that does that. We can do that with a for loop, for i equal to one. So it's going to start out being initialized at one. And while i is less than or equal to pagination total pages. So while it's less than or equal to the number of total pages, let's keep going through and we're going to increment i each time through the loop. Alright, so that gives us a loop.

public function number\_links($url="") {

$output = "";

for($i=1; $i <= $this->total\_pages(); $i++) {

if($i == $this->current\_page) {

$output .= "<span class=\"selected\">{$i}</span>";

} else {

$output .= "<a href=\"{$url}?page={$i}\">{$i}</a>";

}

}

return $output;

}

Now we need our loop to output a URL. So we'll do that with echo a href equals backslash quote backslash quote and then we'll close our tag, and then let's come back and fill in what goes in that in just a moment. Let's see, I just need one of those. Okay, so what text is going to go inside here. We want it to be a number. So that number is going to be i each time through here. So let's use curly braces dollar sign i, and the URL here is going to be similar to the URL we were using for the previous link and the next link.

Do you remember how that worked? We had interpolated the URL that we're setting up here and then after that we have question mark page equals and then we're going to interpolate the page that goes here. And what is that value? Well it's i in this case. Right, we don't have to actually have a previous or next link. We don't have to call a function. It's going to just increment as we go through each one of these. So that should give us a set of links. Let's try it out. Let's go over to our site. And we're on page one.

Let's reload the page. And there we go, one, two, next. I click on next, I have previous one, two, right. And if I click on one, I go back to one. If I click on two, I go back to page two. Now there are some problems here that could use a little bit of cleanup. The first is that we don't have an indication of which one of these is selected. Right now even though I'm on page two, page two is still a link that takes me to page two. So I'm actually just going to add a conditional in here that's going to check and see if we're on the current page or not. So if i is equal to the pagination current page, then we will output something different than when it's not the current page which is when we'll output that link.

So what do we want to output in the case when it's the current page? Well I'm going to put something with span tags around it so it will behave kind of like a link but it's also going to have a class so that we can address it in our CSS. And inside there is still going to be the same thing. We're just going to use dollar sign i. So now it's not going to be a link any more. It's just going to be a span tag that has an i. And the class I'm going to put in here is selected. Alright, so let's go back over here.

Let's reload our page. And now you can see two is no longer a link. If I click on one, now one is no longer a link. So I have some indication of whether they're selected or not. I can improve this a bit further by adding in some CSS. So let's go to our staff CSS real quick. And let's scroll all the way down to the bottom of it. And I've included in the exercise files some very simple CSS called pagination css. I'm going to copy all of that and I'll just paste it in down here at the bottom. So it's just going to take that pagination, add some margin above and below it.

It's going to make sure that we have some spacing on either side of our links and the span tag, and it's going to style the selected span tag to be bold and black. So let's save it, take a look. So now let's reload our page. And we get the new CSS. You might have to force reload to get it to load. So now we have our new CSS and you can see that they're spaced out a little better. That makes it a little bit easier to use. Now we did all of this again just inside index.php. It's not reusable but it's the kind of thing we probably want to reuse.

So let's also take this and let's cut it and let's go to our pagination class. And let's create something else for this. Now we have previous link, we have next link. Let's make another one public function. Let's call this number underscore links. It'll be plural. It'll still take a URL as an argument, just like the other ones did. And then let's paste our code in there. Let me make a couple of modifications to it. Instead of pagination, we're going to use this. So I'll change those.

And we also know that we don't want to echo directly from this. Instead let's do output equals and then each and every time we go through, we'll say dollar sign output equals. And at the very end, of course because it is a function or a method, we need to return back the output. So now we just need to call this number links instead.

So let's go back to index.php and I'll copy this line because it's just going to be very similar, it's just going to be number link.

Alright, see how much cleaner that is? Let's go back and make sure we didn't break anything. We'll reload our page. Click on next. Click on one. Click on two. Alright, it works exactly like we would expect. And now it's inside this function. Now there are also times that we may want all of this all the time. So let's go one step further and let's take our URL and move it up outside of this code. And then we'll take all of this, every single little bit of this, and for now I'm just going to copy it.

Let's go over to the pagination class and let's add one more method here which is going to output all of that for us. We're going to call it just page links. We're passing a URL to it. And what will it do? Well it's going to do that same checking. It's going to check and see whether we need the pagination at all. It's going to output this div and then it's going to output the results of each of those. But again, instead of having this echo back, let's have output equals. And everywhere we have echo, we'll just swap in this append to output line.

There we go. And at the very bottom, we can just return the output and it will do the same things. So now we go back to index.php and all of this code can just become something very simple, page links, passing in the URL and making sure that we echo those results. Look how that works. Nice and simple. Now I'm able to output all of those page links by just calling one simple method on my pagination class.

public function page\_links($url) {

$output = "";

if($this->total\_pages() > 1) {

$output .= "<div class=\"pagination\">";

$output .= $this->previous\_link($url);

$output .= $this->number\_links($url);

$output .= $this->next\_link($url);

$output .= "</div>";

}

return $output;

}

We go back to page one. All of that functionality is now wrapped up in a nice handy method that's inside our pagination class. On index page we have cleaner code like this:

$url = url\_for('/staff/bicycles/index.php');

echo $pagination->page\_links($url);

Now so far we've only paginated the bicycle class. I'll leave it as an exercise for you to do on your own for you to go in and add pagination for your admins and even to go back to the public website and on the inventory of bicycles, to add pagination here as well.

That is not so hard so you can do it on your own to conclude this project!