Course 3: Talking to a MySQL Database in PHP

With <3 from SymfonyCasts

Chapter 1: How to Speak Database

HOW TO SPEAK DATABASE¶

Hey there! Things are about to get crazy because it's time to learn all about **databases**.

We already have a data source that makes our application dynamic. It reads and displays pet data, which happens to be stored in a file called `pets.json`. If we change something in this file, the site updates automatically. In a fully-built site, you'll need to read and write lots of data - like a user's profile, comments, purchases, or maybe forum posts so people can complain about movies. And yea, we *could* build that site by entirely reading data from flat files - like `users.json` and `angry_movie_forum_posts.json`.

What *is* a Database?¶

A database is a place to put data, just like these files. But instead we store things in tables. And we won't be using `file_get_contents` and `file_put_contents` to read and write data, we'll use "queries", which are kind of like human sentences that describe the data you want.

Yea, but what *is* this database thing? So first, a database is a piece of software you run on your computer, just like a web server like Apache. People can talk to the web server by making requests to our machine, usually to port 80. Actually, in our tutorial, we've been using port 8000. A port is like a door and your web server is watching for requests to port 80 so it can process it and return a page.

Talking to a Database¶

To talk to a database, we send a "query" to our computer, usually to port 3306. If the database software is running, it watches for queries coming to this door, interprets them, then sends back the data we're asking for. Yes, we are going to actually do this. But let's focus on the fact that a database is just a standalone piece of software that we talk with to get data.

To make requests to a web server, we typically use a browser. To send a query to a database, we have a few options. But the most basic is to use a command line program called `mysql`. MySQL is the most *common* database software and was installed for you when you installed XAMPP in episode 1.

Tip

Other popular databases are PostgreSQL and SQLite. They're all basically the same, but have subtle pros and cons.

Using the mysql Command Line Program¶

Let's open up the command line. In Mac, I can just type "Terminal" into Spotlight to find it. Oh, and "command line" and terminal mean exactly the same thing. If you're using Windows, we used a terminal inside XAMPP's control panel in episode 1. I recommend using that or downloading Git, which comes with a nice "Git Bash" terminal program. Windows *does* have a built-in terminal called "cmd", but it's really light on features.

Tip

If you are getting a connection error, make sure that MySQL server is running in XAMPP control panel!

We're going to ease into using the command line, so don't worry. You probably *can* avoid using the terminal, but you'll be a much better developer if you and terminal become friendly.

Ready? Type `mysql --help`. Bah! MySQL's help information is a bit chatty. But hey, things are working!

Note

If you get a "command not found" error, make sure that MySQL is installed. If you're using Windows, try using the XAMPP control panel or Git Bash. A lot of things can go wrong during installation, so do your best to search around for a solution for your operating system.

This `mysql` program is like a browser to the database. No, it's *not* the database, just like your browser is not the website. It just helps us *talk* to the database, which might live on some other server.

Connecting to the Database Server¶

If we give it the IP address to the server where the database software is running, it'll let us write queries and send those to it. Instead of typing this all into an address bar, we do it like this:

```
mysql -h localhost --port=3306 -u root -p
```

This says "I want to talk to a database located at the IP address localhost and go through port 3306. You should log into that database using the user root and you should ask me what the password is."

When we hit enter, it asks us for a password. XAMPP gives the root user a blank password, so just hit enter. If that doesn't work, try "root". We're in!

Our login information was just sent to "localhost", which is that special word that points right back to our own machine. It knocked on port 3306. Since XAMPP installed the MySQL database software and configured it to look on this port, MySQL received our details, checked the username and password and basically said, "Welcome, come on in".

Now for Queries!

Chapter 2: Queries, Databases and Tables: Oh My!

QUERIES, DATABASES AND TABLES: OH MY!¶

On this screen, we’re now talking directly to the MySQL database software that’s running on our computer. It’s kind of like talking to a dumb robot, like Siri! We can ask it questions and give it commands, as long as we use a simple syntax it understands.

Let’s try saying hi. Each command should end with a semicolon:

```
Hi Database;
```

Ok, that didn’t work. Try this instead:

```sql
SELECT "Hi Database";
```

Awesome! SELECT is the command we use to get data out of MySQL, and even though this isn’t very impressive, we’ll do crazy things with it later.

And hey, this is our first “query”! The mysql program we’re inside of sent this message out of our computer and back in through port 3306 where the MySQL server software read it and responded back with this little table. A query isn’t a question: it’s a human-like language of commands that say “show me this data”, “update that data” or “delete these things”.

Try another query:

```sql
SHOW DATABASES;
```

I know, using sentences to do things is just weird - it’ll feel better over time. And no, my cat *didn’t* accidentally step on the caps lock - the shouting is on purpose. MySQL commands are usually written in all caps like this, but it’s totally unnecesary - the same command works in lowercase:

```sql
show databases;
```

Creating a Database¶

MySQL is the database software we’re talking to. And actually, it can hold many different databases, which are like separate dividers or folders for data. If you were building 10 different sites, you’d have 10 different databases. I already have several others on my machine.

We’re building a site, so let’s create a database called `air_pup`:

```sql
CREATE DATABASE air_pup;
```

The name `air_pup` isn’t important - we could use anything. Send a query to list the datbases now:

```sql
SHOW DATABASES;
```

There it is! This fancy new database is like an empty directory: there’s nothing in it yet but it’s ready to go!

Hey! We already know 3 MySQL commands, or queries: SELECT, CREATE DATABASE and SHOW DATABASES; And there really aren't that many more to learn.

Creating a Table¶

Let's move into our database:

```
USE air_pup;
```

Any queries we make now will be run against the `air_pup` database.

A directory on your computer holds files. So what lives inside a database? Tables! If you think of a database like a spreadsheet, then each sheet or page is like a table. One sheet might hold pet data and have fields for name, breed, bio and other stuff. We might also have a second sheet that holds user data, with fields like name, address and favorite ice cream. In the database world, each of these is a table.

So before we start tossing pets into the database, we need to create a table and tell MySQL exactly what fields this it will have. We're going to do this the hard way first - you'll thank me later when you *really* understand this stuff!

Like all things, a table is created by asking the database politely. In other words, a query:

```
CREATE TABLE pet(
   id int(11) AUTO_INCREMENT,
   name varchar(255),
   breed varchar (100),
   PRIMARY KEY (id)
) ENGINE=InnoDB;
```

This is long and ugly. First, we say we want to create a table with the name `pet`. Next, like a spreadsheet, we give the table some columns. The big difference is that each column *also* has a data type, which says if it should hold numbers, text or something diferent. The `id` column is an `int` type, so it'll hold numbers.

The `varchar` type means that this column can store up to 255 characters of text. If we try to put more in it, the 256th character will get chopped off!

There are other details that I don't want you to worry about yet. The important parts are that we're calling the table `pets`, giving it 3 columns and setting a data type on each column. Besides `int` and `varchar`, MySQL has a lot of other types. But honestly, you'll use these and just a few others most of the time.

I used multiple lines to make this one long query. MySQL is totally ok with this - it just waits for a semicolon before actually sending the query.

Ok, run it! The message says "Query OK, 0 rows affected". That's not very exciting, considering how much typing we did - but this *is* good!

Try another query to see all the tables in the database:

```
SHOW TABLES;
```

Ok, only 1 table so far, but great start! We've created our database and a table. To celebrate, let's give it a treat by putting some data in it!

Chapter 3: INSERTing and SELECTing Data

INSERTING AND SELECTING DATA¶

How do we add data to a table? With a query! Whenever you want to add a new row to a table, use the `INSERT INTO` query:

```
INSERT INTO pet (name, breed) VALUES ("Chew Barka", "Bichon");
```

Heck, let’s add another one:

```
INSERT INTO pet (name, breed) VALUES ("Spark Pug", "Pug");
```

`INSERT INTO` means “add a new row to this table”. The syntax is a little odd, but always the same: INSERT INTO, the table name, a comma list of the fields you want to fill in, the word VALUES, then a comma list of their values.

SELECTing Data¶

Now that the data is in there, we just need to figure out how to read it. Reading data always starts with the same word, which you’ll see more than *anything* else: SELECT. Try selecting *all* of the data from the `pet` table:

```
SELECT * FROM pet;
```

There they are! And even in a nice table. In this simple form, SELECT shows *every* row and *every* field in a table.

Primary Keys: The very Special id Field¶

Check out that `id` column. Our INSERT query sent values for `name` and `breed`, but not `id`. That’s allowed, and you can even setup a column to have a default value, just for that situation.

But every table has one special column called the primary key. This column is usually an integer that auto-increments. If we don’t send a value for it, MySQL just picks 1, 2, 3, 4 and so on. That’s really handy, because the primary key of each row needs to be unique in the table.

Let’s add another pet, but leave both the `id` and `breed` columns blank:

```
INSERT INTO pet (name) VALUES ("Pico de Gato");
```

Use SELECT to see all 3 rows:

```
SELECT * FROM pet;
```

The `id` column just keeps on auto-incrementing, but this breed is empty. What *are* you Pico do Gato?

SELECT only some Columns¶

A `SELECT` query has a few other tricks too, like being able to return only specific columns instead of everything. Just change the `*` to a list of what you want:

```
SELECT id, name FROM pet;
```

SELECT only some Rows with WHERE¶

One of the most common things to do is filter by some condition. Suppose we *only* want to return rows where the `id` column is *greater* than 1. We do that by adding a `WHERE` clause to the end of the query:

```
SELECT id, name FROM pet WHERE id > 1;
```

You can add filters like this on any column and even use some pretty complex logic. We’ll see more examples later, but since MySQL is so popular, you’ll find plenty of stuff online to help.

Chapter 4: Updating, Deleting and Putting it All Together

UPDATING, DELETING AND PUTTING IT ALL TOGETHER¶

When you’re all done running queries, just type quit.

Now let’s put this all together. MySQL is a database software that runs on a server and listens to port 3306. We tell the `mysql` program that there is database software running on some computer and that it’s waiting for us to talk to it on port 3306. In this case, the database is right on our computer, so we use `localhost` instead of the IP address of some other machine. We also give it a username and password that it uses to log into the MySQL server.

We can actually shorten the command that opens the connection to MySQL:

```
mysql -u root -p
```

We’re removed the `-h` and `--port` options, because the program uses `localhost` and port `3306` by default.

Once we’ve connected to the MySQL server, it contains many databases. We created one, and we can get a full list using the `SHOW DATABASES` query:

```
SHOW DATABASES;
```

To actually make queries to *our* database, we have to use it:

```
USE air_pup;
```

Inside our database, we created one table. We can see *all* our tables using the `SHOW TABLES` query:

```
SHOW TABLES;
```

If you want to see what columns a table has, try the `DESCRIBE` query:

```
DESCRIBE pet;
```

Nice! But here’s a secret: all those commands and queries we just reviewed aren’t really *that* important. Yea, you need to understand them, but you won’t be creating databases or even tables that often. 99% of the time, you’ll be adding, updating, reading or deleting data. So if you want to get dangerous with MySQL, focus on the INSERT, UPDATE, SELECT and DELETE commands.

UPDATE a row¶

Let’s see an UPDATE query in action by capitalizing Spark Pug’s breed. The `id` of his row is 2, so we’ll add a `WHERE` clause to the end of the query so that *only* his row changes:

```
UPDATE pet SET breed='PUG' WHERE id = 2;
```

Check it out by selecting all the rows:

```
SELECT * FROM pet;
```

DELETE a row¶

So actually, you can add a `WHERE` clause to the end of a `SELECT`, `UPDATE` *or* `DELETE` query. Let's remove Pico de Gato by matching on her name:

```
DELETE FROM pet WHERE name = 'Pico de Gato';
```

Yep, looks like that works! So that's basically it! You're now pretty dangerous with MySQL. In a second, we'll get real crazy by talking to MySQL from inside PHP.

PHPMyAdmin: Database GUI¶

So far, I've been making you communicate with MySQL directly using its native language: queries. But there are also some pretty nice GUI's to help you see your data, build queries, and even make tables.

The most popular is probably PHPMyAdmin. It's actually a website, writen in - surprise! - PHP! It runs on your local computer, and if you installed XAMPP, you can already access it by going to `http://localhost/phpmyadmin`:

```
http://localhost/phpmyadmin
```

Oh, and if this doesn't work, make sure that Apache is running. For XAMPP, you can do this in its control panel - we turned Apache off in episode 1, just to prove how we weren't using it for our site.

PHPMyAdmin is easy, and there's plenty of docs online for it. Let's navigate to *our* database and table to check out the data. It even helps us filter the results and show the query that it's using with MySQL behind the scenes.

While we're here, let's add the rest of the columns we need on the `pet` table. These will be the same as what we have in the `pets.json` file, so we'll add `age`, `weight`, `bio` and `image`:

Column Name	Type	Length
age	varchar	255
weight	integer	4
bio	text	
image	varchar	255

Beyond choosing the data type, each column has some other options. These are less important, but you can google them if you're curious.

So now our table is setup and we have an easy way to see and play with our data. Behind the scenes, queries are being sent to our MySQL server software, PHPMyAdmin is just taking care of that for us.

Chapter 5: Talking to Databases in PHP

TALKING TO DATABASES IN PHP¶

If you don’t have the code for the project downloaded yet, get it now. After it downloads, unzip it. Perfect! All we need to do now is start the built-in PHP web server so we can execute our code. Open up a terminal, move into the directory where you just unzipped the code, and start the server:

```
php -S localhost:8000
```

Unless I’ve messed something up, I should be able to go to `http://localhost:8000` and see our site. There it is!

Connecting to MySQL in PHP¶

We’re about to take a *huge* step by talking to our database from *inside* our code. Actually, making queries from PHP is simpler than what we did in the last few chapters. But just like before, step 1 is to connect to the server. Open up `index.php` and create a new PDO object. This shows off a new syntax which we will cover in a second:

```
// index.php

$pdo = new PDO('mysql:dbname=air_pup;host=localhost', 'root', null);
// ...
```

This creates a connection to the server, but doesn’t make any queries. It’s the PHP version of when we typed the first `mysql` command in the terminal to the server.

Before we dissect it, let’s query for our pets! We’re going to use a function called `query` but with a new syntax. Set the result to a `$result` variable. Next, call another function called `fetchAll` and set it to a `$rows` variable. Dump `$rows` so we can check it out:

```
// index.php

$pdo = new PDO('mysql:dbname=air_pup;host=localhost', 'root', null);
$result = $pdo->query('SELECT * FROM pet');
$rows = $result->fetchAll();
var_dump($rows);
// ...
```

Ready to see what the data looks like? Refresh your homepage.

Hey, it’s an array with 2 items: one for each row in the `pet` table. Each item is an associative array with the column names as keys. For convenience, it also repeats each value with an indexed key, but we won’t need that extra stuff, so pretend it’s not there.

What’s really cool is that the array already looks like the one that `get_pets` gives us. If we temporarily comment out that function and use the array from the database by renaming `$rows` to `$pets`, our page should work!!

```
// index.php
// ...
$pdo = new PDO('mysql:dbname=air_pup;host=localhost', 'root', null);
$result = $pdo->query('SELECT * FROM pet');
$pets = $result->fetchAll();

require 'lib/functions.php';

//$pets = get_pets();

// ...
```

Refresh! No errors, and the page shows 2 pets. Sure, they’re not very interesting since our 2 pets don’t have age, weights, bios or images, but we could fill those in easily in PHPMyAdmin.

Chapter 6: Object-Oriented Intro: Classes and Objects

OBJECT-ORIENTED INTRO: CLASSES AND OBJECTS¶

But we just accomplished more than our first query in PHP. We also just saw one of the most fundamentally important concepts to almost any programming language: objects.

To talk to the database, we first open a connection using a *class* called `PDO`. This returns an object, which we set to the `$pdo` variable. We haven't talked about classes or objects yet, but it's not important for you to understand them. Just know that an *object* is another PHP data type. So in addition to strings, numbers, arrays and booleans, we have objects. We'll learn a little about them now, and a lot more in future episodes.

That `PDO` thing is called a class, but you can think of it as a function that has arguments like anything else. Its first argument is a special string that tells it to connect to a MySQL server that's running on our computer and to use the database called `air_pup`. The second and third arguments are the username and password to the server.

But instead of returning an array, boolean or string, this function returns a PDO *object*. Let's dump the variable and refresh:

```
var_dump($pdo);die;
```

This doesn't tell us much, but proves that this is a PDO object. For now, you can think of the words "class" and "object" as meaning the same thing. We'll explore these more in future episodes.

Calling Functions on Objects¶

Over the first few episodes, we've used a lot of functions, including core PHP functions like `array_reverse()` and our own like `get_pets()`. We can call a function from anywhere in our code, so we say that they are available "globally".

But some functions *aren't* global: they live *inside* an object and we call them *through* that object using this arrow (`->`) syntax:

```
$result = $pdo->query('SELECT * FROM pet');
```

Here, `query` is *not* one of those "global" functions: it belongs to the PDO object. If we try to call it like a global function, it doesn't exist!

```
Call to undefined function query()
```

Add the `$pdo->` back so that we're calling a query on that object. Now the page works again.

A Peek into Object-Oriented Programming¶

Classes and objects belong to something called object-oriented programming, which might be the most important programming concept you'll ever learn. But I don't want to get too far into it now. Just be aware that we'll be working with a few objects. And each object has its own set of functions that we can call, like `query` and `fetchAll`.

The Old-School mysql functions¶

I have to warn you: there *is* another way to talk to MySQL in PHP that doesn't use classes and objects. The code looks something like this, and almost every tutorial on the web will teach you this way:

```
mysql_connect('localhost', 'root', null);
mysql_select_db('air_pup');
$result = 'SELECT * FROM pet');
$row = mysql_fetch_array($result);
```

See it? Simple enough? Great. Now, *never* **ever** use this. These functions are old. In fact, they’re said to be “deprecated”, which means that a future version of PHP will remove these. I want you to be a *great* developer, so we’re going to skip straight past these and use the newer, better method and leave this old stuff in the past.

Chapter 7: Centralized Configuration

CENTRALIZED CONFIGURATION¶

The homepage is loading pet data from the database, but our contact page isn't:

> http://localhost:8000/contact.php

It lists a different number of pets than we know we have in the database. Silly contact page. the problem is that `get_pets()` is still reading from the `pets.json` file, instead of using the database.

So instead of putting our query logic right inside of `index.php`, why not just update `get_pets()` to pull from the database?

Copy all of the database-code and paste it into `get_pets()`, which lives in the `lib/functions.php` file. Make sure to return the queried data:

```
// lib/functions.php
function get_pets()
{
    $pdo = new PDO('mysql:dbname=air_pup;host=localhost', 'root', null);
    $result = $pdo->query('SELECT * FROM pet');
    $pets = $result->fetchAll();

    return $pets;
}
```

Try out the contact page! Now that `get_pets()` queries the database, this shows that we have 2 slobbery pets.

Back in `index.php`, remove the database stuff and uncomment out the line that calls `get_pets()`:

```
// index.php
require 'lib/functions.php';

$pets = get_pets();
// ...
```

This works and now all our heavy-lifting lives inside functions. It's not perfect, but our code is getting better!

Creating a Configuration File¶

To make it easier to control your app, configuration - like your database username and password - is usually isolated into its own file. Let's create a new file called `config.php`. Open up PHP and create a new associative array with the databse connection string, the username and the password:

```
// lib/config.php

$config = array(
    'database_dsn'  => 'mysql:dbname=air_pup;host=localhost',
    'database_user' => 'root',
    'database_pass' => null,
);
```

Tip

DSN stands for "data source name" - the fancy name for the connection string.

If we open this file in the browser, nothing happens:

> http://localhost:8000/lib/config.php

But we expected that: the config file doesn't echo anything, it just sets a PHP variable. This file isn't meant to be a page. Instead, we're going to require it from other files and use this `$config` variable.

Let's do this in `get_pets()`. Replace each argument to PDO with a key from the `$config` variable:

```
// lib/functions.php

function get_pets()
{
    require 'config.php';

    $pdo = new PDO(
        $config['database_dsn'],
        $config['database_user'],
        $config['database_pass']
    );

    // ...
}
```

We we refresh, it still works! Remember that `require` is a function that basically copies and pastes the contents of another file into this one. My editor thinks `$config` is undefined, but we know better than that!

Returning from an Included File¶

I don't *love* this. Rename `$config` to `$configVars` in `config.php`:

```
// lib/config.php
$configVars = array(
    'database_dsn'  => 'mysql:dbname=air_pup;host=localhost',
    'database_user' => 'root',
    'database_pass' => null,
);
```

This change *looks* safe. I mean, it's not like we're using this variable anywhere inside this file. But when we refresh, things explode! We *are* referencing the old `$config` variable inside `get_pets`, but that wasn't very obvious.

Remember how we can return values from a function? We can do the same from included files:

```
// lib/config.php
$configVars = array(
    'database_dsn'  => 'mysql:dbname=air_pup;host=localhost',
    'database_user' => 'root',
    'database_pass' => null,
);

return $configVars;
```

Now, instead of relying on whatever we called that variable in `config.php`, create a new variable when you require it:

```php
// lib/functions.php
function get_pets()
{
    $config = require 'config.php';

    $pdo = new PDO(
        $config['database_dsn'],
        $config['database_user'],
        $config['database_pass']
    );

    // ...
}
// ...
```

Try it! It works again. We’re using this file almost like a function: require it and set its return value to a variable. Most included files won’t have a `return` line, but it’s really common for configuration.

So hey, we have a configuration file! The advantage of putting all this stuff into one spot is that you can quickly find and control all the little values that make your app tick. This also makes our app easier to share with another developer. If the database password on their computer is different, they don’t need to dig deep around in your code to find where you hid that. We’re starting to get organized!

Chapter 8: Limiting the Number of Results

LIMITING THE NUMBER OF RESULTS¶

Our app would be a little more fun to play with if we had some more interesting pets in our `pet` table. So I made a file with some SQL queries that insert some new furry friends. Open up the `resources/episode3/pets.sql` file:

```
TRUNCATE TABLE `pet`;

INSERT INTO `pet`
    (`id`, `name`, `breed`, `age`, `weight`, `bio`, `image`)
    VALUES
    (1, 'Chew Barka', 'Bichon', '2 years', 8, 'The park, The pool ...', 'pet1.png');
```

Cool. `TRUNCATE` is the mysql keyword you use when you want to empty a table. It's handy when you're developing - but try not to use this on the production database, you'd probably end up in the dog house. After that, we just have a bunch of INSERT queries. Copy the contents. Now open up PHPMyAdmin, click the SQL tab and paste it in there. And voila! We've got 4 pets.

Tip

You can also execute all the queries in this file from the command line:

```
mysql -u root -p air_pup < resources/episode3/pets.sql
```

When we refresh the homepage, we see our 4 pets. In fact, if we had 1 million pets in our table, they would *all* load on this page. That would be way more dog food than I can afford!

Instead, no matter *how* big this table gets, let's *only* show 3 pets on the homepage. To do this, we can change our query to tell MySQL to only return 3 rows. This is done by adding a `LIMIT` at the end:

```
// lib/functions.php

function get_pets()
{
    // ...

    $result = $pdo->query('SELECT * FROM pet LIMIT 3');
    $pets = $result->fetchAll();

    return $pets;
}
```

When we refresh now, only 3 pets! Hey, that's one more little MySQL trick. But `get_pets()` is kind of broken now - it *always* returns only 3 results. So our contact page says that we only have 3 pets. That's not true!

Instead of hardcoding 3 in the query, let's add a `$limit` argument to the function:

```
// lib/functions.php
function get_pets($limit)
{
    // ...
}
```

Now we can use this to dynamically create the query string. Technically, this is simple. But wait! What I'm about to show you is a *huge* security hole, I mean **huge**! Take the `$limit` variable and add it to the end of the string. This is called concatenation:

```
// lib/functions.php
function get_pets($limit)
{
    // ...

    // THIS IS A HUGE SECURITY FLAW - TODO - WE WILL FIX THIS!
    $result = $pdo->query('SELECT * FROM pet LIMIT '.$limit);

    // ...
}
```

Now, open up `index.php` and pass `3` as an argument to `get_pets()`:

```
// index.php
require 'lib/functions.php';

$pets = get_pets(3);
```

Now refresh. We still have 3 pets!

SQL Injection Attacks¶

But don't celebrate - we have a big security bug! Whenever you create a database query that has some variable part to it, you open yourself up to an SQL injection attack. This is probably the most common attack on the web and one mess-up will give an attacker full access to do *anything* to your databse.

I'll explain this attack more in a few minutes, and we'll close this security hole.

Chapter 9: Optional Function Arguments

OPTIONAL FUNCTION ARGUMENTS¶

But right now, the contact page is broken because `get_pets()` has a required argument, and we're not passing it anything. But what should we pass here? 1000? 1 million? We really want *no* limit, but that's not possible right now.

So let's be clever and pass 0:

```
<!-- contact.php -->
<!-- ... -->

<h1>
    ...your new best friend from over <?php echo count(get_pets(0)); ?> pets.
</h1>
```

Next, add an `if` statement in `get_pets()`. Let's only add the limit if the `$limit` variable is set to something other than zero. We can do this by first creating a `$query` variable. Then, if we have a limit, add a little bit more to the new variable:

```
// lib/functions.php
function get_pets($limit)
{
    // ..

    // THIS IS A HUGE SECURITY FLAW - TODO - WE WILL FIX THIS!
    $query = 'SELECT * FROM pet';
    if ($limit != 0) {
        $query = $query .' LIMIT '.$limit;
    }
    $result = $pdo->query($query);
    // ...

}
```

Try it out! The contact page shows us the right number once again. Our function just got a bit more powerful.

Using Things that Look & Smell Like true/false in an If Statement¶

Whenever we have an `if` statement, we always pass it a boolean: either a variable or little bit of code that equates to `true` or `false`. When we use the `!=` operator, PHP compares the values and feeds either true or false to the `if` statement. That's old news for us.

But let's try something. What if we *just* put the `$limit` variable in the `if` statement:

```
// lib/functions.php
// ...

if ($limit) {
    $query = $query .' LIMIT '.$limit;
}
```

So `$limit` isn't a boolean: it's either the number 3 or 0, depending on who's calling it. So will this work? Will we get an error?

Refresh the contact page. No error - and it shows 4 pets. Now go to the homepage. We see 3 pets. Things are still working!

So you *can* pass something other than `true` or `false` to an `if` statement. And when you do, PHP looks at what's there and tries to figure out if it looks and smells more like true or more like false.

For example, empty things like an empty string or the number zero becomes false. Everything else becomes true.

Value	Equates to
0	false
empty string	false
empty array	false
null	false
false	false
5	true
foo	true
array(5, 10)	true
-25	true
true	true

Even a negative number or a string that has only spaces in it becomes `true`, because these aren't really empty. So when we pass `0` for the `$limit` argument, it looks like `false` and doesn't enter the `if` statement.

Optional Argument¶

But we can go further and make the `$limit` argument completely optional. How? Just assign it a value when you declare the argument in `get_pets()`:

```
function get_pets($limit = 0)
{
    // ...
}
```

This doesn't automatically change anything. But now we can remove the argument we're passing to `get_pets()` in `contact.php`:

```
<!-- contact.php -->
<!-- ... -->

<h1>
    ...your new best friend from over <?php echo count(get_pets()); ?> pets.
</h1>
```

Prove that it works by going to the contact page.

The Nothing Value: null¶

Let's change the `0` default value to `null`. `null` is the value that is equal to "nothing", and I used it earlier as my database password, since my MySQL server doesn't have a password.

Both 0 and `null` will look like `false` in the `if` statement, so the code will act the same. Since `null` means nothing, it just feels a little bit better to use it as the default value for an optional argument.

The Function Signature¶

By the way, the name and arguments to a function are called the "signature". When you hear people talking about a function's signature, it's just a smart-sounding way to refer to the arguments that function has and also the value it returns. So now you'll sound even smarter when talking with your cool new programmer friends.

Chapter 10: Using Query Parameters

USING QUERY PARAMETERS¶

It’s time to give each pet their very own page. At this stage, a new page means a new file - so create a `show.php` file and copy in the require statements for `functions.php`, the header and the footer:

```
<!-- show.php -->
<?php require 'lib/functions.php'; ?>
<?php require 'layout/header.php'; ?>

<h1>Display 1 pet</h1>

<?php require 'layout/footer.php'; ?>
```

Tip

Eventually, we’ll learn a strategy called “routing” that makes creating new pages easier and gives you tons of control over how your URLs look.

Every page follows a familiar pattern: we do some work at the top - like querying the database - and then we use the variables we created in the rest of the HTML to print things. Here, our “work” will be to query for only *one* pet in the database.

But first, let’s create a link from each pet on the homepage to this file. To tell the page *which* pet we want to display, let’s add `?id=` to the end of the URL and print out this pet’s `id`:

```
<!-- index.php -->
<!-- ... -->

<h2>
    <a href="/show.php?id=<?php echo $cutePet['id'] ?>">
        <?php echo $cutePet['name']; ?>
    </a>
</h2>
<!-- ... -->
```

Refresh and click on the link. We’re taken to the new page with a little `?id=` part on the URL. That’s called a query parameter and it’s the easiest way to pass some extra data to a page.

Using Query Parameters¶

The HTTP request coming into the server now contains a little extra information via this query parameter. So how can we read this in PHP? Whenever you need some data from the incoming request, the answer is *always* one of those superglobal variables. We used `$_POST` to get data submitted in a form and `$_SERVER` to figure out if this is a GET or POST request.

Query parameters are accessed via the `$_GET` superglobal. Let’s dump this whole variable:

```
<?php
// show.php
var_dump($_GET);die;
```

Just like the other superglobals, this is an associative array. We can add more values in the URL by adding an & sign between each:

> http://localhost:8000/show.php?id=5&foo=bar&page=10

Now `$_GET` has 3 items in it.

Tip

Query parameters always start with a `?` and then every key-value pair is separated by an `&` afterwards.

So let's grab the id key from `$_GET`:

```php
<?php
// show.php
require 'lib/functions.php';
$id = $_GET['id'];
// ...
```

You know the drill from here! We'll query the database for this one pet and use that information to build the page. But wait! Should we build the query right here or put it in a function? Easy choice: a function will help keep things organized. Call an imaginary `get_pet()` function and pass it the `$id`:

```php
<?php
// show.php
require 'lib/functions.php';
$id = $_GET['id'];
$pet = get_pet($id);
// ..
```

And before we even think about creating that function, we already know what it will return: an associative array with the details for just *one* pet. Let's build out this page with that in mind. To save some typing, I've started this file in the code download at `resources/episode3/show.php`. I'll copy its contents into this middle of our page and fill in a few missing pieces:

```
<!-- show.php -->
<!-- ... -->

<h1>Meet <?php echo $pet['name']; ?></h1>

<div class="container">
    <div class="row">
        <div class="col-xs-3 pet-list-item">
            <img src="/images/<?php echo $pet['image'] ?>" class="pull-left img-rounded" />
        </div>
        <div class="col-xs-6">
            <p>
                <?php echo $pet['bio']; ?>
            </p>

            <table class="table">
                <tbody>
                    <tr>
                        <th>Breed</th>
                        <td><?php echo $pet['breed']; ?></td>
                    </tr>
                    <tr>
                        <th>Age</th>
                        <td><?php echo $pet['age']; ?></td>
                    </tr>
                    <tr>
                        <th>Weight</th>
                        <td><?php echo $pet['weight']; ?></td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>
```

Chapter 11: Querying for One Pet

QUERYING FOR ONE PET¶

Ok, now to create that `get_pet()` function! Add it in `functions.php`:

```
// lib/functions.php
// ...
```

First, let's build the query. Instead of returning *every* row in a table, we can use the WHERE clause trick we learned earlier to return *only* the row whose id equals the `$id` variable argument. Like before, this query has a variable part to it, so it **is a security** flaw. But we're going to fix it in a few seconds:

```
function get_pet($id)
{
    $query = 'SELECT * FROM pet WHERE id = '.$id;
}
```

Understanding Function Scope¶

Now we have the query, but we *don't* have the `$pdo` object. It's created and lives in `get_pets()`, but we can't access it here. Each function is its own little universe and you only have access to the arguments passed in and any variables you create in that function.

Let's see what I mean. Create a `$test` variable outside of the `get_pet()` function and then try to dump it inside of it:

```
// lib/functions.php
// ...

$test = 'works?';
function get_pet($id)
{
    var_dump($test);die;
    $query = 'SELECT * FROM pet WHERE id = '.$id;
}
```

Yep, that explodes! The variable `$test` does not exist. Like I said, each function is its own, isolated universe. When you talk about what variables you have access to and why, we often use the word "scope". If I say something about the "function scope", I'm talking about all of the variables that my function has access to. It's just another one of those techy terms that really describe something simple. Now we can remove this test variable.

We know that copying and pasting the code from `get_pets()` into `get_pet()` is code duplication and a recipe for future problems. So how can we get the `$pdo` variable in `get_pet()`?

One answer is to make it accessible anywhere by moving that stuff to a new function called `get_connection`. Let's do that:

```
function get_connection()
{
    $config = require 'config.php';

    $pdo = new PDO(
        $config['database_dsn'],
        $config['database_user'],
        $config['database_pass']
    );

    return $pdo;
}
```

With variables, you have to worry about scope and what you have access to. But functions can be called from anywhere. Call this function from `get_pets()`:

```
function get_pets($limit = null)
{
    $pdo = get_connection();
    // ... all the same logic
}
```

Now, use it again from inside `get_pet()`. To finish this, use the same `query()` function we saw before. But at the end, use `fetch()` instead of `fetchAll()`:

```
function get_pet($id)
{
    $pdo = get_connection();
    $query = 'SELECT * FROM pet WHERE id = '.$id;
    $result = $pdo->query($query);

    return $result->fetch();
}
```

Use `fetchAll` to return multiple rows from the database and use `fetch` when you just want *one* row, like here.

Refresh! It works! I never doubted you. This looks good, but we *have* to address our big security hole next.

Chapter 12: Preventing SQL Injection Attacks with Prepared Statements

PREVENTING SQL INJECTION ATTACKS WITH PREPARED STATEMENTS¶

Both `get_pets` and `get_pet` contain an SQL query where one part of it is a variable. Whenever you have this situation, you're opening yourself up for an SQL injection attack. Want to see one in action?

Change the id value in the URL of your browser to a very specific string:

> http://localhost:8000/show.php?id=4;TRUNCATE pet

Hmm, so things look ok. But refresh again. The pet is gone! In fact, check out the database in PHPMyAdmin - the `pet` table is empty! Seriously, we just emptied the *entire* pet table by playing with the URL!

Let's dump the query in `get_pet` and refresh:

```
function get_pet($id)
{
    $pdo = get_connection();
    $query = 'SELECT * FROM pet WHERE id = '.$id;
    var_dump($query);die;
    $result = $pdo->query($query);

    return $result->fetch();
}
```

This is an SQL injection attack:

> SELECT * FROM pet WHERE id = 4;TRUNCATE pet

By cleverly changing the URL, our code is now sending *2* queries to the database: one to select some data and another to empty our table. If our site were in production, imagine all the dogs that would never get walked and cats that would never get fed salmon. All because someone injected some SQL to drop all of our data and put us out of business. These attacks can also be used to silently steal data instead of destroying it.

Fortunately, we can get our data back by re-running the queries from `resources/episode3/pets.sql` in PHPMyAdmin.

Save our Site¶

How do we save our site? With prepared statements. Don't worry, it's an unfriendly term for a simple idea. Prepared statements let us build a query where the variable parts are kept separate from the rest of the query. By doing this, MySQL is able to make sure that the variable parts don't include any nasty SQL code.

Using Prepared Statements¶

To use prepared statements, our code needs a little rework. First, change the `query` function to `prepare` and put a `:idVal` where the id value should go. This returns a `PDOStatement` object, so let's rename the variable too.

Next, call `bindParam` to tell MySQL that when you say `:idVal`, you really mean the value of the `$id` variable. To actually run the query, call `execute()`:

```
// lib/functions.php
// ...
function get_pet($id)
{
    $pdo = get_connection();
    $query = 'SELECT * FROM pet WHERE id = :idVal';
    $stmt = $pdo->prepare($query);
    $stmt->bindParam('idVal', $id);
    $stmt->execute();

    return $stmt->fetch();
}
```

It's a bit longer, but it works! We built the query, but replaced the dynamic part with a placeholder: a word that starts with a colon. This isn't any PHP magic, it's just how prepared statements work. The `prepare()` function does *not* actually execute a query, it just returns a `PDOStatement` object that's ready to go. To actually make the query, we call execute, but not before calling `bindParam` for each placeholder with the real value. To get the data from the query, we finish up by calling either `fetch()` or `fetchAll()` .

This is *actually* how I recommend you write all your queries, whether you have any variable parts of not.

Let's repeat the change in `get_pets()` . Just write the query with the LIMIT placeholder, bind the parameter, call `execute()` to make the query and `fetchAll()` to get the data back. Simple!

```
// lib/functions.php
function get_pets($limit = null)
{
    $pdo = get_connection();

    $query = 'SELECT * FROM pet';
    if ($limit) {
        $query = $query .' LIMIT :resultLimit';
    }
    $stmt = $pdo->prepare($query);
    $stmt->bindParam('resultLimit', $limit);
    $stmt->execute();
    $pets = $stmt->fetchAll();

    return $pets;
}
```

Refresh the homepage to make sure it's working. Oh crap, it's not! But there's also no error.

Debugging Errors¶

Whenever there's a problem with a query, we can configure PHP to tell us about it, instead of staying silent like it is now. Find `get_connection()` and add one extra line to configure our PDO object:

```
function get_connection()
{
    // ...

    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    return $pdo;
}
```

Refresh now! Ah, a nice error! So this is what it looks like if your query has an error. The problem is actually subtle and really not that important. Pass a third argument to `bindParam` :

```
$stmt->bindParam('resultLimit', $limit, PDO::PARAM_INT);
```

Now it works. This tells MySQL that this value is an integer, not a string. This almost never matters, but it does with LIMIT statements. So like I said before, don’t give this too much thought.

When you *do* have errors with a query, the best way to debug is to try the query out first in PHPMyAdmin, then move it to PHP when you have it perfect.

Moving On!¶

Ok team, we are *killing* it. In just a few short chapters, we’ve updated almost our entire application to use a database. The only part that *doesn’t* use the database is the new pet form, which we’ll fix early in the next episode.

Use your new-found power for good: create some tables in PHPMyAdmin and start querying for data. Don’t forget to put up your fence to protect against SQL Injection attacks, so that the adorable poodle, fluffy Truncate Table, doesn’t cause you problems.

Seeya next time!