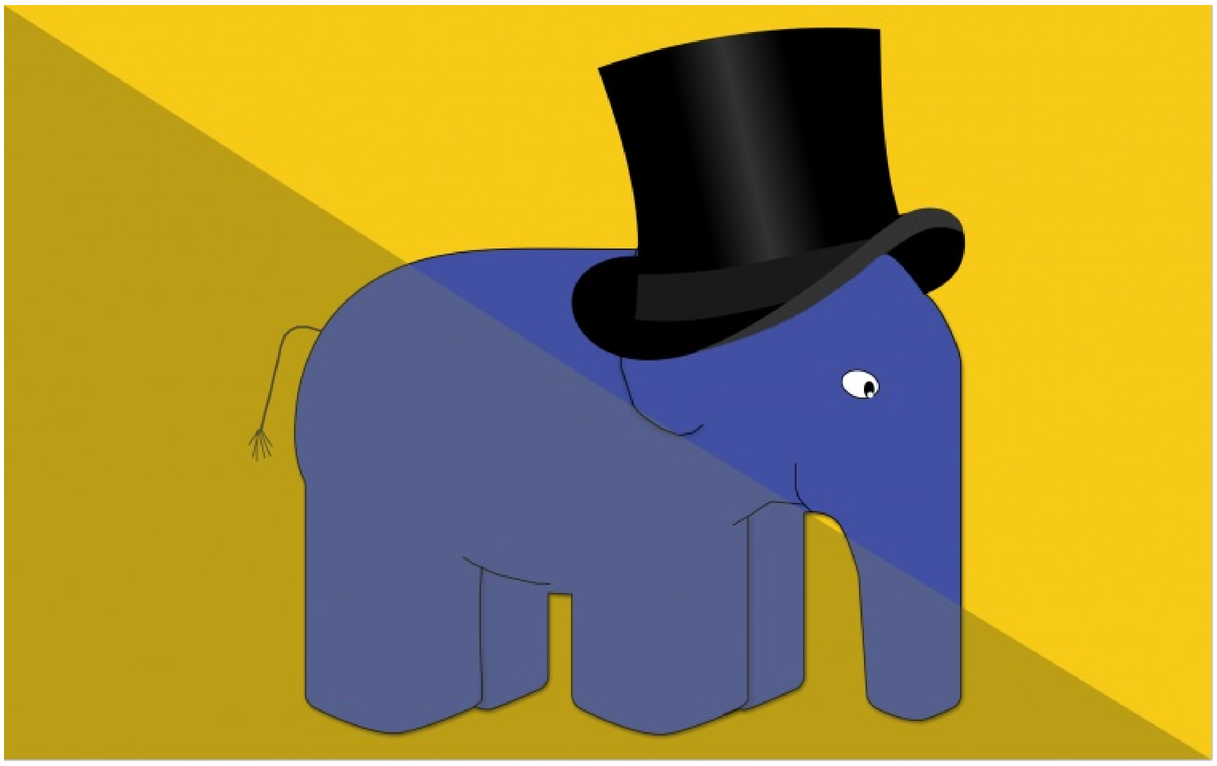
Course 2: How to stop worrying & start writing PHP



With <3 from SymfonyCasts

Chapter 1: Request, New Page and - Hey, You Welcome Back!

REQUEST, NEW PAGE AND - HEY, YOU WELCOME BACK!¶

Hi there! I knew you'd be back! Learning to be a PHP programmer is a lot of work, but totally worth it. So, keep going: you're getting more dangerous with each minute you spend practicing.

Getting the Code, Starting the Web Server¶

We're going to keep coding on the project from episode 1. If you don't have that code, just download it from the screencast page and unzip it. Once you've done that, we just need to start the PHP web server.

Open a terminal and move into the directory where the unzipped files live. I already unzipped the files in a `Sites/php` directory. Start the PHP web server by typing the following:

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

Great! Now just put that URL in your browser and voila!

> http://localhost:8000

Tip

Having issues? Check out our server setup chapter in episode 1!

What Secrets lie on the HTTP Request?¶

In episode 1, we learned that the web works via requests and responses. Our browser sends an HTTP request message into the interwebs for airpup.com/contact.php. This eventually finds our server where it scratches on the door. And with any luck, some web server software like Apache will be listening, open the door, look in a certain directory for the `contact.php` file and process all the PHP treats . . . I mean tags. The final HTML is called an HTTP response, and is sent from the server back to our browser.

So it all starts when *we* send that HTTP request message. But actually, that message has a whole lot more information than just the hostname and page we want. It also has our IP address, info on which browser we're using and the values of fields when we submit a form.

Peeking at Requests¶

Let's peek at these by going to the debugger on our browser. Click on the network tab and refresh.

Tip

You may need to enable network tracking before refreshing, which the network tab will ask you to do. If you don't see any debugging tools, make sure you're using a good browser. I recommend Google Chrome.

Each line here is an HTTP request that was just made by our browser. The top is the request for `contact.php`. The other requests are for the CSS, JS and images on the page. Yep, on every page load, your browser is actually making a *bunch* of requests into the internet tubes.

If you click on `contact.php`, we can actually see how *this* HTTP request message looks. Yep, it's a lot of stuff. That `User-Agent` is what browser you're using and `Accept-Language` is how the browser tells the server what languages you speak. It's not important now, but we can get any of this information from inside PHP!

We need a New Page!¶

I want to build a form so a user can rent out their pet's love to others.

Let's start by creating a new page. How do we do that? Just create a new file called `pets_new.php` and scream some text out:

```
HI I AM A PAGE!

SQUIRREL!
```

To see us screaming at us, change the URL to `/pets_new.php`:

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

So every new page is just a new file. In a future episode, I'll teach you a way to create URLs and pages that's much fancier than this. But don't worry about that quite yet.

Let's make this page properly fancy by copying in the `header.php` and `footer.php` require lines:

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

HI I AM A PAGE!

SQUIRREL!

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

Refresh to see a page that is only a little ugly. Progress! And a little Twitter Bootstrap markup makes this look a tiny bit better.

```
<div class="container">
    <div class="row">
        <div class="col-xs-6">
            <h1>Add your Pet</h1>
        </div>
    </div>
</div>
```

Navigation like a Real Site¶

Hmm, and wouldn't it be nice if we had a link to this page from our top menu? Let's change "About" to say "Post" and link to this page. This code lives in `header.php`. We can also make "Home" *actually* go to the homepage.

```
<!-- layout/header.php -->

<ul class="nav navbar-nav">
    <li class="active"><a href="/">Home</a></li>
    <li><a href="/pets_new.php">Post</a></li>
    <li><a href="#contact">Contact</a></li>
    ...
</ul>
```

Refresh. Go team! We have a working nav like a real site!

Chapter 2: We deserve to Create a Form

WE DESERVE TO CREATE A FORM¶

Ok, let's build this form! Let's leave PHP alone for a second and put together some good ol' HTML to create a form with name, breed and weight text fields. And hey, why not a bio textarea field too so we can get to know these pets!

```
<h1>Add your Pet</h1>

<div class="form-group">
    <label for="pet-name" class="control-label">Pet Name</label>
    <input type="text" name="name" id="pet-name" class="form-control" />
</div>
<div class="form-group">
    <label for="pet-breed" class="control-label">Breed</label>
    <input type="text" name="breed" id="pet-breed" class="form-control" />
</div>
<div class="form-group">
    <label for="pet-weight" class="control-label">Weight</label>
    <input type="number" name="weight" id="pet-weight" class="form-control" />
</div>
<div class="form-group">
    <label for="pet-bio" class="control-label">Pet Bio</label>
    <textarea name="bio" id="pet-bio" class="form-control"></textarea>
</div>

<!-- ... -->
```

I'm adding some divs and classes here that use the Twitter Bootstrap CSS that's included in this project. So, nothing to worry about - this just makes our site a little prettier! And hey, that's important too!

Note

I used `<input type="number" ... />` for the `weight` field because newer browsers will render this with some extra nice-ness. Old browsers just render a normal text fields. This is an HTML5 field.

In PHP's eyes, the important thing is the `name` attribute on each field. It can be anything, but this will be how we get each field's value in PHP.

We also need to wrap all of the fields in a `form` tag. Set its `action` attribute to point back to this same page and definitely don't forget the `method="POST"` part. I'll show you why that's so important in a second:

```
<h1>Add your Pet</h1>

<form action="/pets_new.php" method="POST">
    <!-- all the form guts from before ... -->
</form>

<!-- ... -->
```

Finally, you're going to need a cool-looking submit button in the form!

```
<button type="submit" class="btn btn-primary">
    <span class="glyphicon glyphicon-heart"></span> Add
</button>
```

Refresh. Wow! Now that's a sweet form.

Submit that Form¶

Ok, let's fill out a few fields and submit.

Hmm, nothing happens. The URL is still the same, and now the form is blank again. Actually, something amazing just happened. Go back to the network tab in our debug tools. When we pressed “Add”, our browser sent an HTTP request to the server. And this time, it had even *more* information on it. It had all the values from our form!

GET and POST Requests¶

It’s different in another way too. The “Request Method” is POST - and that’s because of the `method="POST"` part of our form tag. So there are 2 *types* of HTTP requests our browser can send. GET request are used on almost every page and POST requests are used on most form submits. The difference doesn’t really matter yet, but GET requests are for reading data, like when you click on a link and read the next page. POST requests are for sending data, like a form submit.

Tip

There are actually other HTTP methods like PUT, HEAD and DELETE. These are mostly useful only if you’re building or using an API. Don’t worry about that at all right now.

So when we submit, our browser sends a POST request that has all the values on it. If we could just read that, we’d be as dangerous as a loose puppy at a hot dog stand!

Chapter 3: Reading POST’ed (Form) Data

READING POST’ED (FORM) DATA¶

That’s done with a super-magic variable called `$_POST`.

Try it! Let’s use my favorite debugging tool `var_dump` on this variable:

```
<?php var_dump($_POST); ?>
```

Fill out the form again and submit. It prints out an associative array. Each key is from the `name` attribute of the field and its value is what we typed in!

Where does $_POST Come From?¶

Wait, not so fast! Where did this variable come from? We already learned that if we try to reference a variable name that doesn’t exist, PHP gets really angry and tells us about it:

```
<?php var_dump($_POST); ?>
<?php var_dump($fakeVariable); ?>
```

Like when we try this, we *do* get a big warning.

Here’s the secret: `$_POST` is one of just a few variables called “superglobals”. That’s nothing more than a heroic way of saying that `$_POST` is always magically available and equal to any submitted form data.

The other superglobals include `$_GET`, `$_SERVER` and a few others. We’ll talk about them later, but they all have one thing in common, besides their love of capital letters. Each super-global gives you information about the HTTP request our browser sent.

Getting the Form Data¶

Eventually, we’re doing going to save this new pet somewhere, like a database. We’ll get there, but for now, let’s set each value to a variable and dump them to prove it’s working. I’ll do this right at the top of the page, because that’s a nice place to put your PHP logic:

```
<?php
$name = $_POST['name'];
$breed = $_POST['breed'];
$weight = $_POST['weight'];
$bio = $_POST['bio'];

var_dump($name, $breed, $weight, $bio);die;
?>

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

This time, refresh your browser. This actually re-submits our form with the same data we entered a second ago. And there it is!

I took advantage of a cool thing about the `var_dump` function: it accepts an unlimited number of arguments. Most functions accept 0, 1, 2 or more arguments. That’s normal. But a few brave guys accept an unlimited number. `var_dump` is one of those brave functions, and it’s documentation shows us that.

$_POST on GET Requests¶

Click the top nav link to go to the homepage, and then click “Post” to come back to this form. Oh no, we’re blowing up! When we clicked the link, our browser sent a simple GET request to the server. We didn’t submit a form to get here, so `$_POST` is an *empty* array. This means our keys don’t exist, and PHP is giving us a lot of ugly warnings about it!

Coding Defensively¶

Let’s fix those warnings first. Any time you reference a key on an array, you gotta ask yourself: Is it possible that this key might ever *not* exist? It’s better to plan for these cases then let your site have big errors later. Always think about how your code *could* break and code so that it doesn’t.

Right now, this means we need to add some `if` statements to see if the array keys exist, starting with the `name`:

```php
<?php
if (array_key_exists('name', $_POST)) {
    $name = $_POST['name'];
} else {
    $name = 'A dog without a name';
}
```

We can shorten this slighty by using `isset`. It’s just like `array_key_exists`, but shorter and a bit easier to read:

```php
<?php
if (isset($_POST['name'])) {
    $name = $_POST['name'];
} else {
    $name = 'A dog without a name';
}
```

Repeat this for all of the fields:

```php
if (isset($_POST['breed'])) {
    $breed = $_POST['breed'];
} else {
    $breed = '';
}

if (isset($_POST['weight'])) {
    $weight = $_POST['weight'];
} else {
    $weight = '';
}

if (isset($_POST['bio'])) {
    $bio = $_POST['bio'];
} else {
    $bio = '';
}

var_dump($name, $breed, $weight, $bio);die;
```

Refresh! Ok, warnings are all gone. But we still need to be smarter. When we make a normal GET request, I don’t want to bother looking for any form data, I just want to render the HTML form. I really only want to run all of this logic when the browser sends a POST request, meaning we *actually* just submitted the form.

Detecting GET and POST Requests: $_SERVER¶

So how can we find out if our code is handling a GET request or a POST request?

If you’re thinking the answer is in one of those superglobal variables, you nailed it! This time, it’s `$_SERVER`. Let’s dump it out to see what it looks like:

```php
var_dump($_SERVER);die;
```

Woh! It’s an associative array, and it has a *ton* of stuff in it, 25 things in my case. What is this stuff? Well, it’s information about the HTTP request that was just sent. See the `HTTP_USER_AGENT` key? That comes from a piece of information our browser included in the request.

No, you don’t need to memorize this, or really remember any of it. Occasionally you’ll need some information, like the user agent. And when you google for how to get that in PHP, this will be your answer.

See that `REQUEST_METHOD` key? Ah ha! That’s the HTTP method, which is GET right now.

Let’s wrap all of our form-processing logic in an `if` statement that checks to see if the `REQUEST_METHOD` key is equal to `POST`:

Refresh! Our browser makes a normal GET request. All that form processing stuff is skipped and we got our normal, beautiful HTML form. And when we fill out the form and submit, our browser sends a POST request. Now our code kicks into action and dumps out all that data. We’re not *doing* anything with our form data yet, but our workflow is looking good!

Chapter 4: Saving Pets

SAVING PETS¶

Ok, let's talk about saving pets. No, not like *rescuing* them, though that's really cool too. I'm talking about being able to submit our new pet form and saving that pet information somewhere so that it shows up on our site. I want to make our pet list truly dynamic!

We haven't talked about databases yet, and we're not using one. But actually, the pet data our site needs *is* being stored in a simple `pets.json` file. And this file *is* something we can read from and even update. And hey, that's basically all a database really does. So if we can figure out how to update the `pets.json` file each time we submit this form, we're in business!

First, we can re-use our `get_pets` function to get an array of all of the *existing* pets from the file. Let's add this right at the bottom of our form processing code:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    // ...

    $pets = get_pets();
    var_dump($name, $breed, $weight, $bio);die;
}
```

Next, create an associative array that represents the new pet that's being added. Make sure the keys you're using match the existing pets from the file. We don't have age or image fields yet, so just set those to be blank:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    // ...

    $pets = get_pets();

    $newPet = array(
        'name' => $name,
        'breed' => $breed,
        'weight' => $weight,
        'bio' => $bio,
        'image' => '',
        'age' => '',
    );

    var_dump($name, $breed, $weight, $bio);die;
}
```

Ok! Now just add that new pet to the `$pets` array:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    // ...

    $pets = get_pets();

    $newPet = array(
        'name' => $name,
        'breed' => $breed,
        'weight' => $weight,
        'bio' => $bio,
        'image' => '',
        'age' => '',
    );

    $pets[] = $newPet;

    var_dump($name, $breed, $weight, $bio);die;
}
```

Remember that the empty square brackets tells PHP we want to add something to the `$pets` array, but we don't care what the items key is. It'll choose a unique number for us.

Saving the Pets to pets.json¶

Now, `$pets` has all the existing little fur balls, *and* our new one. It basically represents *what* we want to save to `pets.json`.

Let's do it! First, turn `$pets` back into JSON with PHP's `json_encode` function. To *actually* save the file, use another PHP function: `file_put_contents`:

```
// ...
$pets[] = $newPet;

$json = json_encode($pets);
file_put_contents('data/pets.json', $json);
```

This is basically what the `get_pets` function does, only in reverse! `json_encode` turns the array into a string, and then we save it back to the file.

Let's try it! Fill out the form and submit. An error!

> Call to undefined function get_pets()

Ah, woops! That function lives in the `functions.php` file. If we want to use it, we need to `require` that file:

```
<?php
require 'lib/functions.php';

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // ...
}
```

Ok, refresh and re-post the form. Hmm, it looks like it did nothing. But that's not true! We submitted the form, our code detected this was a POST request, we saved the new stuff to `pets.json`, and then the page continued rendering the blank form. There weren't any fireworks, but I think this worked!

Go to the homepage to find out for sure! We didn't give it an image, but there's our pet. We don't even have a database, and we already have a dynamic app.

Readable JSON!¶

If you look at `pets.json`, it got flattened onto one line. That's ok! Spaces and new lines aren't important in JSON, and PHP saved without any extra whitespace. Again, that's fine really.

But since I *did* like my file better when it was readable, give `json_encode` a second argument of `JSON_PRETTY_PRINT`:

```
$json = json_encode($pets, JSON_PRETTY_PRINT);
```

Fill out our form again. Hey, now `pets.json` looks awesome again. We are really good at training this digital pet :) `JSON_PRETTY_PRINT` is called a *constant*, which is kind of like a variable, exept that it’s magically available everywhere, doesn’t have a `$`, and its value can’t change. You won’t use them often, so don’t worry about them too much.

Chapter 5: The Art of Redirecting

THE ART OF REDIRECTING¶

Go back to the new pet form and fill it out again. Ok, it looks like it’s still working. Refresh a few times and check out `pets.json`. Woh, we have a lot of duplicate “Fidos” in our file! Each time I refreshed, the form resubmitted and added my pet *again*. Bad dog!

In the real world, we don’t want users to be able to create duplicate records accidentally or so easily. And that’s why you should redirect the user to a different page after handling a form submit.

Remember: we *always* send back an HTTP response to the user. And so far, a response is *always* an HTML page. But it could be something else, like a tiny bit of directions that tell the browser to go to a totally different URL. The browser would then make a *second* request to that URL and display *that* HTML page. This is called a redirect.

Returning a Redirect Response¶

After saving the new pet, let’s tell the user’s browser to redirect to the homepage. To do this, use a function called `header` and pass it a string with the word `Location`, a colon, then the URL. Finish up with our trusty `die` statement:

```
// ...

$json = json_encode($pets, JSON_PRETTY_PRINT);
file_put_contents('data/pets.json', $json);

header('Location: /index.php');
die;
```

And sure enough, this time when we fill out the form, we’re redirected to the homepage! Check out the network tab again in our debugging tools. It shows us the request for the current page like always, *and* the request of the last form submit.

Redirect: 2 Requests/Responses¶

Remember, 2 request-response cycles just happened all at once. When we submitted the form, a POST request was sent, which we can see. But the response that our PHP code sent back to the browser didn’t contain HTML. Actually, it didn’t contain *anything*, except for this little `Location` line that told the browser to redirect to the homepage.

When the browser sees this instruction line instead of HTML, it quickly makes a GET request to the homepage. This time, our code returns a response message with HTML and it displays it. It looked instant, but now we know that our browser just made 2 separate requests.

Headers¶

Let’s learn something that takes most web developers *years* to figure out. Ready?

Don’t crowd the elevator doors when it opens, people might be getting out of it.

Ok, want to learn something else that usually takes web developers years?

When our browser makes a request, the most important part is the URL. Of course! The server needs to know which page we want! But the request also has *other* information like our IP address and browser details. Each extra bit if info is called a request *header*. And we can read these in PHP from that `$_SERVER` array variable.

The response our code sends back *also* has extra information, called *response* headers. A response is basically 2 pieces: the HTML and these headers. Most of the time, we don’t think about headers or responses: we just write HTML and print some variables. This automatically becomes the content of the response and a few important headers are set for us.

But sometimes, you *do* need to send back a response with a bit of extra information. And in fact, when you want to tell a browser to redirect, we need to send back a response message with a `Location` header. This type of extra information is added to the response with the `header` function and each has the same format: a header name, a colon, then the value. Every browser is programmed to look for the `Location` header.

After, I put a `die` statement just to stop everything right there. We haven’t printed anything yet, so the response has no content. That’s perfect: I want the browser to go somewhere else, not display this page. But even if we did echo some HTML, the user would never see it because the browser would redirect so quickly.

I thought you said die was bad?¶

Ok, I admit, in episode 1, I said that you should never use `die` except for debugging. Yes, I’m violating that temporarily because we need to learn a few more things before we can re-organize code and get rid of this. We’ll see that in a future screencast.

Can’t Re-Submit: Mission Accomplished¶

We started all this redirect business because we didn’t want the user to be able to refresh a finished form and create duplicate pet data. And now, we’ve done that! Refresh here: instead of re-submitting the form, it just makes another GET request to the homepage. Whenever you process a form submit, add a redirect by setting the `Location` response header.

Chapter 6: Cleaning up with save_pets

CLEANING UP WITH SAVE_PETS¶

Hey, I have an idea: let's clean up our code a little by creating a `save_pets` function in `functions.php`:

```
function save_pets()
{

}
```

Copy in the 2 lines that encode the pets array and writes the file:

```
function save_pets()
{
    $json = json_encode($pets, JSON_PRETTY_PRINT);
    file_put_contents('data/pets.json', $json);
}
```

Ah, my editor is highlighting the `$pets` variable because it's undefined. A function only has access to the variables you create *inside* of it, and, yep, there's definitely no `$pets` variable here. It lives back in `pets_new.php`.

Adding Arguments to Our Functions¶

We already know that functions can have arguments - we just saw that with `header`, which has one argument. We can make `save_pets` require an argument too - just add a `$petsToSave` variable between the parenthesis of that function:

```
function save_pets($petsToSave)
{
    $json = json_encode($pets, JSON_PRETTY_PRINT);
    file_put_contents('data/pets.json', $json);
}
```

Now, call the function from `pets_new.php`. When we do, PHP will *require* us to pass it an argument. Give it the `$pets` variable:

```
$pets[] = $newPet;

save_pets($pets);

header('Location: /');
die;
```

Accessing an Argument from inside a Function¶

Ok, now dump `$petsToSave` so we can see what's going on:

```
function save_pets($petsToSave)
{
    var_dump($petsToSave);die;
    $json = json_encode($pets, JSON_PRETTY_PRINT);
    file_put_contents('data/pets.json', $json);
}
```

Fill out the form. Bam! We see the big pets array. Change the argument to `save_pets` to just a little bit of text:

```
$pets[] = $newPet;

save_pets('this is some text!');

header('Location: /');
die;
```

Refresh! Now our text is dumped out. So whatever we pass to `save_pets` becomes the `$petsToSave` variable. Change the argument back to `$pets`:

```
$pets[] = $newPet;

save_pets($pets);

header('Location: /');
die;
```

Inside `save_pets`, we can pass `$petsToSave` into `json_encode`. Oh, and I just invented this variable name - we could have called it anything:

```
function save_pets($petsToSave)
{
    $json = json_encode($petsToSave, JSON_PRETTY_PRINT);
    file_put_contents('data/pets.json', $json);
}
```

Moment of truth! Refresh. We're back on the homepage with the pet we just added. Brilliant!

Why Use Functions?¶

Things work the same as before, so why did I make you add this function? Moving logic into functions gives us 2 really cool things.

First, if we need to save pets somewhere else, we can just re-use this function. We're already doing this with `get_pets`, which I think we're calling in at least 3 places.

Second, by moving these lines into a function with a name, it helps explain *what* they do. If I didn't write this code, I might have trouble figuring out what it does or its purpose. But when it's inside a function called `save_pets`, that helps.

We'll go through this process of writing code and reorganizaing it over and over again. Our site is measured by more than just whether or not it works: you also want your code to be easy to understand and easy to make more changes too.

Building great things *and* doing it well, *that's* where you're going. And you've just finished another episode you crazy developer! Congrats! Now build something and keep learning with us.

Seeya next time!