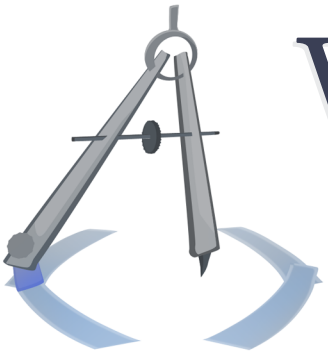

Website Architecture

Lezione 10

HTML Forms

Finally, an explanation as to how to accept data from the user through Web forms, looking at client-side and server-side technologies.

Michael Serritella

Summer 2010

Intro to HTML Forms

HTML forms are a bridge between client-side and server-side technologies. They are a collection of simple input widgets whose data are sent to the server in a GET or POST request. Server-side languages like PHP can access these variables conveniently.

HTML elements

The HTML elements used for forms were quite awkwardly designed. Most of the controls use the same tag - `<input>` - and vary only by their `type` attribute. The others have rather ridiculous names, in that their names or their meanings are nearly unguessable if you only know one or the other.

Here is one example:

```HTML
<!-- A text box -->
First Name: <input type="text" name="FirstName" value="(First name)">
```

Basic structure

Alas, they all have something in common. The `name` attribute of the element is used for the variable name as it will appear in the server-side program. And the `value` attribute, if it exists, is used for the variable's value. If an element has a `value`, then usually it is the initial value before any user input and it may be changed.

Form controls may have attributes that specify behavior, like the maximum number of allowed characters, and they have some antiquated ones that specify width and height, et. al.

Form controls are wrapped in a `<form>` element, which specifies the type of HTTP request, the URI of the processing page, and an optional name or ID for the form that helps to identify it within the DOM. A page may have multiple forms, but forms may not be nested.

Basic example

Here is an example of a basic form:

HTML

```
<form method="POST" action="processingPage.php">
  First Name: <input type="text" name="FirstName" value="(First name)">
  Last Name: <input type="text" name="LameName" value="(Last name)">
  <br>

  <input type="radio" name="Gender" value="M" checked> Male
  <input type="radio" name="Gender" value="F"> Female
  <br>

  <!-- This 'checked' is a "minimized attribute". In XHTML these are
       invalid and must be written as 'checked="checked"', which is
       also valid for HTML. -->
  <input type="checkbox" name="IsCitizen" value="1" checked> Citizen?
  <br>
  <br>

  <!-- The value is actually the visible text for buttons. -->
  <input type="submit" value="Submit!">
</form>
```


<input> elements

Here are the common variations on the `<input>` element; each of these is a value for `type`.

text A single-line text box.

password A single-line text box whose text is hidden (usually via ***).

checkbox A check box, whose `value` is the value if it is checked. The 'checked' minimized attribute may be used here.

radio A radio button box, whose `value` is the value if it is "checked". The 'checked' minimized attribute may be used here. Radio buttons in the same logical grouping will share a `name` attribute.

button A button which does not inherently do anything; it may be used for JavaScript events. Its `value` is its visible text.

submit A submit button. Its `value` is its visible text.

image An image which serves as a submit button. Its `src` is its URI, and there is also an `alt`.

reset An reset button, which resets the form controls to their initial values. Its `value` is its visible text.

hidden An opportunity to write a name & value pair which does not appear in the rendering.

Other elements

These are the element names of the other common form elements:

textarea A multi-line text box; the opening and closing tag surround the default text.

select A drop-down menu, in almost all cases, though it may be rendered as any other kind of chooser. May also be a multi-selector via the 'mulitple' minimized attribute.

option The choices of a `<select>` tag; they appear as children of that tag. They may have a `value` which is different from their displayed text. If no `value` is given, the contents of the tag will be the displayed text and the submitted value. If the tag has contents and a `value`, then the contents are for display only and the `value` is submitted. This tag has a 'selected' minimized attribute.

Documentation

See the **standard** or **W3Schools** for information on their extra options and quirks.

Form labels

There is also a `<label>` tag which can assist the other form elements. It appears as a text label, and when clicked, it can activate an associated control. For a check box, this can mean that if you click on the text label, it toggles the check box.

The text of the label is the contents of the `<label>` tag. You can associate a label with a control in two ways. One: you give an ID to the control and then provide the ID as the `for` attribute of the label. Two: you encompass the form control in the label tag. In this case, you can type the label's text anywhere as a sibling of the form control, but it might be nice to wrap it in an anonymous `<span>` just so you don't have CSS problems.

Example:

HTML

```
<input id="CitizenCheckbox" type="checkbox" name="IsCitizen" value="1">
<label for="CitizenCheckbox">U.S. Citizen?</label>

<!-- Alternate structure: -->
<label>
  <span>Male</span>
  <input type="radio" name="Gender" value="M" checked>
</label>
```


Auto-complete

Modern browsers give the user the option of saving form data so that it may be re-populated on subsequent visits to the form. For instance, every time you go to the Gmail login page, the browser gives you a list of possible usernames which pops up as you start typing. If you want to disable this, then you can modify the form control's HTML element and set the attribute `autocomplete` to "off".

But.. this is browser-dependent. Can you think of a method which is browser-independent? It may require an extra dash of cleverness on your form rendering and processing pages.

PHP data structures

So far, so good. Good enough. Now, how to access this data via PHP? It's actually quite easy - use some built-in superglobal arrays.

Superglobals for form inputs

In PHP, a superglobal is a variable which is global in all scopes; you do not need the `global` declaration or anything like that. There are a few such superglobals which are very useful in receiving form data. They are `$_GET` and `$_POST`. These are associative arrays which simply contain the name => value pairs of each form control. Not bad!

What is sometimes vexing is that only the form controls with nonempty values will be sent - at least with most browsers - and/or they are the only ones for which variables will be built within `$_GET` and `$_POST`. In your production code, if you are unsure whether a value will come through, you can just check the value with an if-statement or use `isset()`. When debugging, you can use the function `print_r()` or `var_dump()`

to see which form controls came through. Note that you can always walk through these values using `array_keys()`.

Arrays of form controls

PHP allows for some syntactic sugar here.. Exploiting the `[]` syntax of pushing array elements, you can actually give your form controls *names that end in this operator*, and they will be built into an array within `$_GET` or `$_POST`.

Example:

HTML

```
<input id="CitizenCheckbox" type="checkbox" name="IsCitizen[]" value="1">
<label for="CitizenCheckbox">U.S. Citizen?</label>

First Name: <input name="FirstName[]" value="">

<!-- Multiple controls with the same name are fine when you
     use this construct; they will appear as IsCitizen[0],
     IsCitizen[1], etc. -->
<input id="CitizenCheckbox" type="checkbox" name="IsCitizen[]" value="1">
<label for="CitizenCheckbox">U.S. Citizen?</label>

First Name: <input name="FirstName[]" value="">
```


Warning: register_globals

Since the creators of PHP are so sharp, they made this feature wherein the form variables were *injected directly into global scope*. This is obviously a security hole. So, since then, they have made it variable by a configuration option, which is basically always turned off. The option is called register_ globals and is documented **here**.

You will not need to use register_globals, but you need to be aware that it exists. Since it still exists in the configuration options, if some hacker gets access to your configuration, he/she could switch this to on and then cause other problems. Simply for this reason - that it could conceivably be switched on - you should know it exists.

Form validation

Here comes the ugly part. Sort of. You should probably care whether the input is anything close to your expectations. You should check it out. This process is called form validation, and it is basically a thorn in the side of computer science. It's not too hard, but it's annoying in that there is probably no great solution that is not too restrictive in other ways. Anyhow, let's see our options.

Client-side

In client-side form validation, you use JavaScript to check out the values of the form elements. Their DOM objects will have a **`value`** property, which you can use to get or set the value. They also have a boolean **`disabled`** property that temporarily turns off the control's interactivity[1].

But how do you catch the submission event before it happens? Simple; listen on the **submit** event, possibly with an **`onsubmit`** attribute. Use the "return HandlerCall();" syntax to give your-

[1]FYI: They have a `disabled` HTML attribute that works the same way.

self the option of cancelling the event and thus stopping the form submission.

Example:

HTML

```
<form method="POST" action="asdf.php" onsubmit="return FormIsValid();">
  <!-- Elements -->
</form>
```

Within JavaScript, you will need a way to access the form elements via the DOM. You can give them all IDs, though that may be cumbersome. You can give the form an ID, but if you do, you should also give it a name (using **name**) which is the same thing. This is due to old quirks. Now, you can hopefully retrieve the form via the ID or through the **document.forms** array, as **documented by the Mozilla Developer Center**. You could also pass the **this** variable to the handler from within your **onsubmit** attribute.

Server-side

And now for something completely different. You can submit all the data to the server and then validate it from within PHP. If the form is invalid, you can do any number of typical things: write an error page and then redirect back after 5 seconds; display the form again with errors highlighted; lock the user out; etc.

Re-rendering the form is probably most common and most prudent. You can do this in a number of ways. Here are a few common options:

- Write the form-rendering code in a function which both scripts can access. The function takes some information via a parameter as to which elements were invalid so that it can render them differently.
- The form display and form processing pages are the same script, and there is an if-statement in the script that decides which page content it will display. If the form is invalid, it will render the form again, but it will render it differently since it may want to show which controls were invalid.

Once you have decided which form controls are invalid, you can give them an extra CSS class, like InvalidFormControl. Then,

in the rule definition for InvalidFormControl, you can give a red background, red border, dark red text, etc. If the user has started to change the input (detect via JavaScript), maybe change the class to an intermediate class like InvalidFormControlChanged, which has colors that are halfway normal.

Security issues

So, which do you do? Client-side or server-side validation? Unfortunately, you usually end up doing a little of both. You should at least do server-side validation, but there may be reasons why you would still want to catch it on the client side.

First, let's justify our choice. Why server-side? Well, a clever client doesn't have to go through your form in order to send data to your processing page; they can *GET or POST from anywhere*. So the data may be wildly different from what you expect. Thus, the only sure way to validate a form is server-side.

But if we do client-side, does that buy us anything? Well, it may save us a hell of a lot of traffic. Probably 95% of errors

will be honest mistakes, and client-side validation is OK for honest mistakes. You could be saving spurious traffic costs on that 95%. Especially if your connection is encrypted with SSL, those transfer costs may add up; they may hog your RAM and a significant chunk of your maximum allowable connections. So, unfortunately, client-side validation has a place.

Fortunately, it does not need to be fully correct. Your server-side implementation must be correct, but you can do approximate or incomplete validation on the client side. If you want to be very economical, you can validate only the form controls for which you expect the bulk of user error, and you may only check against one common case of misuse. It's up to you.

Client-side validation is usually a tedious process. If your site is not too concerned with traffic limitations and you are pressed for time, you can forego client-side validation entirely.

Uploading files

You can, of course, contribute to the degradation of humanity and actually upload a file to the Internet. This is done with familiar HTML but with some unique handling by PHP.

HTML element

The HTML element goes as follows:

```HTML
<input type="file" name="FileVariableName"> Tada!
```

This writes a filename text box and a "Browse" button next to each other; they come as a package deal. To stylize this, you need some voodoo. See the **page on quirksmode**.

Variable names and request details

The request should be POST. Some antiquated browsers apparently support a PUT request, which is not really appropriate for typical Web servers. Use POST. The `name` of the

element is the name of the variable as it will appear in PHP's arrays (which we will see soon). The file(s) must be preceded by a (probably hidden) element named "MAX_FILE_SIZE", which gives a maximum number of bytes. Yes, it's not reliable in and of itself for security. But you can still set a server-side maximum via PHP's configuration options.

Basic process in PHP

In PHP, you will want to use the `$_FILES` superglobal. The documentation actually has a nice section on the subject **here**. Basically, you find an entry in `$_FILES` which corresponds to your form control's name. This entry is also an array, and it has some metadata, including the size, type, and file's original name. The file is initially put in a temporary folder and given a temporary name. You can then choose to move it somewhere permanent.

Sending email in PHP

PHP includes a very simple mechanism by which you can send email, assuming the server allows it, which is virtually always the case on paid servers.

Basic example

Use the `mail()` function, documented **here**.

Example:

PHP

```php
<?php
mail("Joe@Whatever.biz", "Hey", "Hey, would you like to blah");
?>
```

Header syntax

If you want to send more information than the basic set shown above, like a (spoofed) sender, you have to write additional headers. These headers will actually have a very familiar syntax, since HTTP headers are ostensibly modeled from them.

Same style:

```
Header-Name-actually-case-insensitive: Header value
```

So, you just need to build a string like this, and then you can pass it to `mail()` as a fourth argument. If you want to send multiple headers, you have to make one string which has all of the headers you want, separated by newlines. Just like HTTP, there is a blank space between these headers and the message body. If you want to send something fancy, like attachments or HTML email, then check the documentation for `mail()` and read the comments.

Security issues

If you are accepting user input for any part of the email, like for the recipient address, you may be vulnerable to an attack. A hacker may use your form to send spam!

Remember that the header structure of email is just like that of HTTP; there are headers, and then a blank line, and then the message body. If you are accepting some part of the mail head-

ers (e.g. recipient address) as user input, then the user could *inject a newline* and change the structure of the outgoing message. He/she can write additional headers, write a blank line, and write a new message body, all the while pushing down the text of your original headers and original body. They may appear as gibberish text that is appended to the spam message's body.

To avoid this, simply check the input for newlines. You may think that you should strip the message of newlines or do whatever translation, but really, if the message contains newlines, then it's certainly dishonest, so you should probably just not send the message. Log the activity if you want.