Keyword Buttons Development Environment

I must be honest. I developed my own coding environment, not because I felt I could do a better job then what was currently available, but really because there were so many tools out there, I didn't know where to begin, and many of them had a fairly steep learning curve – at least those environments or languages that could handle the "Application", that I envisioned many years ago.

The application I envisioned was Keyword Buttons, hereafter referred to as the *application*.

In truth, my "application", had not been fully defined and scoped out. Which is why I developed around the "unknown". That is, I wanted my development environment to be able to pivot, in as many ways as possible, around a rather "global" construct. Such as assignment or setting of variables or other "language" features using well-established technologies of today, and likely those of tomorrow…

HTML, CSS, JavaScript/JQuery, and other language constructs where parameters to the corresponding code designations, are delineated with a substitute-able variable, can be asserted, server side by a search and replace function, on the user level using a system of scripts.

In my case I developed a system and method in PHP that demonstrates these ideas.

Writing my system in this way took longer, but the experience has been worth it in terms of what was learned along the way.

Code - a Common Thread

A common thread in coding webpages, is that one thing is "set" to another. For example the src= parameter of an img tag, a color: value, in CSS, or even a function name in JavaScript or JQuery. Class names are set, and "blocks of code" are set.

All of these things can also be set programmatically using PHP, and assigning the variable portion, by the PHP system, using, and assigning $$ variables (embedded in templates) and writing them out as a webpage, and then reloading the page in its modified form. All assuming the variable aspects of the page are known and valid values are substituted in, as appropriate.

On top of that I envisioned, what I called a "Content Reference". The "Content Reference", in my environment, can be defined as a url, that usually, but not always, has an "extension" associated with it, embedded within a public area (web-exposed) website. Some examples of content references would be.

(a link to a PDF file)

(a link to an Image file)

(a link to a YouTube Video)

(a link to a search engine)

(a link to an application or API)

Content References (showing related content)

In particular a Content Reference is a url that yields “a *known type* of end point data” that can be read by a scripting engine without requiring an API or other authentication procedure, in which to access the url, independent from the site owner. In my case, Content References are located using “dynamic urls”, which are invoked programmatically.

Dynamic urls are simply pre-defined search urls (a url template) with the variable portion for the search assigned programmatically.

This is similar to a Search Engine, which identifies public urls, by spidering sites using common methods that are well established, and then redisplays these links in their search engines domain (Google, Bing, etc), or the search engine of the “source domain”.

The difference being the Search Engine Page of the site is targeted as opposed to the domain itself – there is no parsing through robots.txt.

The end point data usually refers to data that is encapsulated in a single file, or of a single “common type”. The type of file is an abstraction, but usually, helps to limit the scope of contents of the content reference.

That is for example, the scope of a .jpg file has a target of an img tag, and similarly, the scope of a PDF file, is code on a page that accepts as a parameter a PDF file or url, a PDF viewer, for example.

Further to this analogy, a content reference file that is a video, that will be able to be embedded in video controls, be it an HTML5 code chunk, or another type of code that accepts video urls as a parameter to its embodiment.

And commenting out code can be represented by specific $$vars, as established by the application, and corresponding to the syntax of a given comment statement, with respect to the language imparted.

That being said, from this approach we can build on the ideas of a content reference, not only for using content we own, but for content that is publicly available by linking to it.

In fact, this definition enabled a great test bed for proof of concept.

The concepts described here are that of associating large groups of keywords with large groups of corresponding content urls, so that additional processing can be done after having isolated and assembled into a cogent system urls that are categorized by content type (and user).

Content References (continued)

One such organizational structure is like this.

User

Topic

Keyword

Content References

Images (identified with image types)

Videos (identified with video types)

Articles (identified with article types)

Specialized (identified with website)

Application (corresponds to a keyword application)

The Application Screen

During my walk down this road, I discovered that part of the complication with websites, including mobile sites, is the way people have solved the issues of rendering content on various devices with varying screen sizes.

For me, this seemed to be complicating my purposes, because of the technologies required to write a site that renders both on a mobile device, and on a “regular” computer screen, which isn’t really regular either (the resolutions and physical dimensions can vary). Media tags, grids, frameworks, etc. have addressed the issue somewhat, but what I decided on was to simplify the rendering to a fixed block, that could fit into any size screen (if the zoom factor was modified accordingly, by the end user, or programmatically).

In part, I decided that I would leave it up to the device owner to set their zoom settings, that best fits their device, and let the browser handle rendering the site at a workable level – and keep the content of the site at a minimum to support this type of design.

The zoom features in modern browsers have a great ability to render the sites with excellent quality, at zoom percentages, that suit most any device.

Page Rendering (User Level)

Since the application requires a number of screens (known and unknown), I decided on a naming convention for the files that constitute the front end.

So for example a login screen might be data.dyna.login.html, where .dyna indicates this is a webpage that is dynamically rendered (for example an IMG tag is populated with a Content Reference url), and the target of the submit button on the webpage, might be populated dynamically, for example the action parameter filled in with https://fleshner.com/process.login.php?user=$$user, where the $$user is dynamically changed, to the current user…

> *All and all, one aspect of naming things the way I did was to provide for an easier method of handling things programmatically, that is if I wanted to access a series of pages, then having some sort of systematic method, of naming and rending webpages (via a primary page) was paramount.*

In the end, I decided that the index.php page would actually render a unique page for each and every end user, and let the server side PHP handle filling in the variable portions of the page. The variable portions of the pages could be $$variables embedded in a src= parameter, or embedded in class assignment (for a given tag), or even in inline CSS or other CSS, depending on the application requirements, as described earlier.

Security – a Diamond in the Rough

> A great bonus/caveat of rendering user level pages, instead of static pages that support many users, is security. Since I have full control, at the moment the page is created (for a given access instance) I can do whatever I want to it, without effecting *other end users*, including dynamically applying, for example other encryption algorithms, persistency, and an ability to further restrict mischievous bots or other security hacks from being introduced into the system, because they are dynamic points on the web, moving changing, and thus not sitting ducks.

Main Points

The Keyword Buttons Project (The Application)

1. Constrain Exposed Content (limit the dimensions of the webpage throughout the site).
2. Identify Pages with naming conventions (.dyna pages or processor pages such as PHP).
3. Identify Variable Sections with an identifier (prefixed with $ or sometimes $$).
4. Use Search and Replace Functions when Rendering Pages
5. Embed Variables in CSS
6. Use a common include (for PHP functions)
7. Embed Variables in HTML tags
8. Isolate Pages as User Level (or guest content)
9. PHP (render HTML as PHP)
10. Domain Integration (use full paths)
11. Apache Webserver (using Named based Virtual Domains)
12. Define a “Content Reference”
13. Define a “Dynamic Url”
14. Indicate Exceptions (where this process does not work).

Some of the main points numbered above were not addressed in this document, however the import of them will become apparent when using the site: http://fleshner.com

http://fleshner.com/about 03/27/2021 v1.2