DESIGNING HTML HELPERS TO OPTIMIZE WEB APPLICATION DEVELOPMENT

Dragos-Paul Pop[1]

Abstract

Building a web application or a website can become difficult, just because so many technologies are involved. Generally companies tend to people that work in teams to develop web applications. These teams are made up of professionals that focus on different technologies, such as CGI, HTML, JavaScript, CSS and databases. When the work of many people gathers to make up a single document there is often a mismatch between parts of code written by different team members. This article focuses on improving this matter by bringing consistency in code through the use of HTML helpers in server-side scripting languages. The examples in this article use PHP as the server-side language, but the model can be applied in any other language a developer works with.

Keywords: HTML, CGI, Helper, OOP, code generation

JEL Classification: L86

Introduction

From its invention around 20 years ago, the World Wide Web has known one of the highest ascensions of all computer technologies. Most jobs put on the market by companies are web development related and most IT students tend to go the "web way". We must admit that web development has somewhat of a bigger attraction than other IT technologies, mainly because it is easier to learn and lets people be more creative. Little effort is required to create a web page and most students are attracted by this fact. But there is a catch. Web development includes quite a lot of technologies and although getting started is easy, the learning curve gets steeper once people need to develop more complex websites or applications.

When speaking of "the web" we include a lot of technologies. Web applications follow a client-server model and the World Wide Web is not a proprietary system. Actually, the system is just a way of interlinking hypertext documents and accessing them via the Internet. Several protocols have been design to accomplish this and there is a governing body that develops standards for the web. This body is known as the World Wide Web Consortium (or W3C) and is not a company, but a standards organization. This means that its goal is not profit. The W3C is made up of 317 organizations and includes all the great IT companies and corporations, like Microsoft, Apple, Mozilla, Yahoo, Google, IBM, HP and so on. The W3C is responsible of many web standards, like (X)HTML, CSS, XML and so on.

[1] Ph.D. Student at the Academy of Economic Studies and Teaching Assistant at the Romanian-American University, Bucharest, Romania; email: dragos_paul_pop@yahoo.com

I mentioned earlier that the web follows a client-server model. The W3C is responsible for the most important standards that technologies running on the client side should follow, like HTML, CSS, XML and the DOM, to name some of the most important. There are other client side technologies that are not governed by W3C like JavaScript, Flash and Silverlight. JavaScript follows the ECMA Script standard ruled by Ecma International. Flash is developed by Adobe and Silverlight by Microsoft.

All the client side technologies are used to build hypertext documents that are interpreted by a web browser and presented to the end user. This is also an application that runs on the client machine. Browsers are diverse and are built by various companies and organizations. Microsoft makes the infamous Internet Explorer. Mozilla makes Firefox. Apple builds Safari and Google develops Chrome. We also have Opera, Konqueror and many more others. These are the most popular browsers people use on the Internet today.

There is also a server side to the web application model. This is where the documents and all the information are stored. There are a lot of technologies for the server side too. These are used to store, access, fetch and build all the documents and information needed by the client. The application that handles document storage, retrieval and access is the server software. There are many server applications used today, but two of them stand out as the most used: the Apache Web server and Microsoft's Internet Information Services. On top of file management functions, these applications allow running scripts. This is where the CGI technology comes in. It is used to make websites and web applications more dynamic. It actually gives a developer a way to build hypertext documents on the fly, based on what information the user requested. These technologies allow data to be stored in databases. Scripting technologies include PHP, ASP, Java, Perl, Python, Ruby, Cold Fusion and a lot more.

The problem

Most hypertext documents residing on the server are dynamic. That means they will actually go through a CGI application on the server before being fed to the client. This allows for programming logic to be inserted in these documents. Often times, these documents, or scripts, tend to contain a mix of different types of code: HTML, CSS, JavaScript and CGI. CSS and JavaScript code can be taken out of the main document and put into different files, but server-side scripting code cannot. This leaves developers with files that contain two types of code.

CGI languages give an option to require and include code from other files. This gives a way of separating logic from content even further. Still, a lot of the static HTML code needs to be generated by CGI, like tables, forms, lists and so on. This becomes a burden on the CGI developer, because he needs to focus on two technologies and have two types of code in his files. It also becomes a burden for the client side developer, because he needs to work with HTML code generated by the CGI developer. Many times table, form, list and other markup generated by the CGI programmer tends to be inconsistent across pages.

A typical portion of a server-side scripting file looks like the code snippet below. It generates a list of items that are used as a navigation menu.

```
<ul style="border: thin slid black">
<?php  if(isset($_SESSION['user'])) echo "<li><a
href='logout.php'>LOGOUT</a></li>";  ?>
<?php  if(isset($_SESSION['user'])) echo "<li><a
href='cont.php'>MY ACCOUNT</a></li>";
else echo "<li><a href='login.php'>LOGIN</a></li>"; ?>
<li><a href="cart.php" onclick=”goto(‘wherever’); return
false;”>BASKET</a></li>
<li><a href="contact.php">CONTACT</a></li>
<li><a href="galerie.php">GALLERY</a></li>
<li><a href="faq.php">FAQ</a></li>
<li><a href="despre.php">ABOUT</a></li>
</ul>
```

We can see that HTML, PHP, JavaScript and CSS code is intertwined. This style of programming gives headaches.

The solution

A way of solving the above problem is to build special pieces of code (mainly classes, for CGI languages that allow an object oriented programming style) that will be used to generate markup code. This way, the markup is no longer a burden on either the server-side developer or the client-side developer.

On the client side, JavaScript provides a way of doing just this by using special methods and interacting with the DOM (document object model). This means that developers can use JavaScript to build and insert or modify HTML elements. It is as easy as calling the createElement function and passing it the type of HTML element to be built. The function returns a reference to a newly built object and then properties of it can be modified. The newly created element can then be inserted anywhere in the document. Similarly an existing element can be fetched and modified.

Server-side scripting languages don’t usually have functions that provide this functionality. But it can be built by the developer.

The HTMLElement class

The proposed model borrows heavily from JavaScript to help the developer build HTML elements and work with them. It is provides methods for creating elements, adding, removing and changing attributes, inner code and child nodes. Basically it is a DOM manipulation class.

The listing of the class follows bellow with a detailed explanation of each method defined.

```
class HtmlElement {
```

The class properties or attributes are:

- the element itself (this is a string and represents the name of the HTML tag; egg.: a, p, div, form, table etc.)
- a collection of attributes for the element (this is an array with key-value pairs representing the attribute name and attribute value; egg.: id-element1, class-error etc.)
- the inner HTML value of the element that is found between the opening and closing tags of the element; this could be plain text or HTML
- a collection of children of the current element; this is an array of other instances of this class
- the parent of the current element; represents an instance of the same class; crucial for DOM manipulation purposes
- finally the HTML code representation of the element, its attributes and children

```
public $element = "";
private $attributes = array();
public $innerHTML = "";
public $children = array();
public $parent = NULL;
private $code = "";
```

The __construct function is the class constructor and takes none or one or two arguments: the element (the tag) and an array of initial attributes.

```
public function __construct($element="", $attributes = NULL) {
  if(!empty($element))
    $this->element = $element;
  if(!empty($attributes))
    $this->attributes = $attributes;

}
```

The four functions bellow are part of PHP's magic methods. These are used to create and manage dynamic properties for a class instance. There are getters and setter for dynamic properties, as well as a destroyer and a checker. All of them work with the *attributes* property of the class, adding, fetching, checking or destroying certain key-value pairs.

```
public function __get($attribute) {
  if (array_key_exists($attribute, $this->attributes))
        return $this->attributes[$attribute];
}

public function __set($attribute, $value) {
  $this->attributes[$attribute] = $value;
}

public function __unset($attribute) {
  if (array_key_exists($attribute, $this->attributes))
    unset($this->attributes[$attribute]);
```

```
}

public function __isset($attribute) {
  return array_key_exists($attribute, $this->attributes);
}
```

A usage example for the above methods is as follows:

```
//creating a new instance of the class
$paragraph = new HTMLElement();
//setting a known property
$paragraph->element = "p";
//dynamically adding a first property
$paragraph->id = "first_paragraph";
//dynamically adding a second property
$paragraph->align = "left";
```

The next method creates the HTML code representation of the object. Basically it creates the *code* property. It begins with the opening tag, adds the attributes and, if the element has a closing tag, adds the children code and the closing tag.

```
private function make_code() {
  if($this!=NULL) {
    $code ="<";
    $code .= $this->element;
    $code .= $this->add_attributes_code();
    if($this->end_tag()) {
      $code .= ">";//\n";
      $code .= $this->innerHTML;
      $code .= $this->add_children_code();
      $code .= "</";
      $code .= $this->element;
      $code .= ">";//\n";
    }
    else
      $code .= " />";//\n";
    $this->code = full_trim($code);
  }
  else
    $this->code = '';
}
```

The *add_child* method receives an instance of this class and adds it as a child to the current element, by appending it at the back of the *children* array. Also, the child element gets its parent set to the current element to provide DOM accessibility.

```
public function add_child($child) {
  if ($child instanceof HtmlElement) {
    $this->children[] = $child;
    $child->parent = $this;
  }
}
```

For better DOM manipulation there are two more methods for adding children. The *add_child_after*, that adds an instance of the class after a certain child that has the id property provided as the first argument, and *prepend_child* that does what the name suggests.

```
  public function add_child_after($neighbour_id, $child) {
    if($child instanceof HtmlElement) {
      foreach ($this->children as $key=>$kid) {
        if($kid->id == $neighbour_id) {
          array_splice(
$this->children, $key, 1, array($kid, $child)
);
        }
      }
    }
  }

  public function prepend_child($child) {
    if ($child instanceof HtmlElement) {
      array_unshift($this->children,$child);
    }
  }
```

The following function generates a string that represents the attributes of the current element and their values.

```
  private function add_attributes_code() {
    $code = "";
    if(!empty($this->attributes))
      foreach ($this->attributes as $attr=>$val)
        $code .= ' '.trim($attr).'="'.trim($val).'" ';
    return $code;
  }
```

An important method is found just below. It provides a way of generating the HTML representation of the children of the current element. It actually calls each child's own *make_code* method and appends the string to the return value.

```
  private function add_children_code() {
    $code = "";
    foreach($this->children as $child) {
      $code .= $child->code();
    }
    return $code;
  }
```

The *code* method provides a more accessible and friendly way of accessing the element's *code* attribute.

```
  public function code() {
```

```
    $this->make_code();
    return full_trim($this->code);

  }
```

The next method return true or false depending on whether the element has an ending tag or not. It just uses a switch statement based on W3C HTML standard tags.

```
  private function end_tag() {
    switch ($this->element) {
      case "img" :
      case "input" :
      case "area" :
      case "base" :
      case "br" :
      case "col" :
      case "frame" :
      case "hr" :
      case "link" :
      case "meta" :
      case "param" :
        return false;
    }
    return true;
  }
```

The last methods are DOM accessing methods. They are used to get references to certain children of the current elements. The developer can search for a child by its ID or by a certain property-value pair. References to multiple children can also be returned by searching for a certain tag or common property-value pairs.

```
  public function get_child_by_id($id) {
    if($this->id == $id)
      return $this;
    foreach ($this->children as $child)
      if($el = $child->get_child_by_id($id))
        return $el;
    return false;
  }

  public function get_children_by_property($prop, $val) {
    $res = array();
    if($this->$prop == $val) {
      return $this;
    }
    foreach ($this->children as $child) {
      $el = $child->get_children_by_property($prop, $val);
      if(is_object($el))
        array_push($res,$el);
      elseif(!empty($el))
        foreach($el as $e)
          array_push($res,$e);
```

```
    }
    return $res;
  }

  public function get_child_by_property($prop, $val) {
    $kids = HtmlElement::get_children_by_property($prop, $val);
    if($n = count($kids))
      return $kids[$n-1];
    else
      return NULL;
  }

  public function get_children_by_tag_name($tag) {
    $res = array();
    if($this->element == $tag) {
      return $this;
    }
    foreach ($this->children as $child) {
      $el = $child->get_children_by_tag_name($tag);
      if(is_object($el))
        array_push($res,$el);
      elseif(!empty($el))
        foreach($el as $e)
          array_push($res,$e);
    }
    return $res;
  }
}
```

Basic usage of the class is given in the examples bellow. A new HTML Form element is created and attributes are set. Then several other elements are created and used to populate the form.

```
$form = new Form();
$form->enctype="multipart/form-data";
$form->class = "edit_avatar";
$form->method = "post";
$form->action = "cont/edit/avatar";
$fieldset = new HtmlElement("fieldset");
$legend = new HtmlElement("legend");
$legend->innerHTML = "Change avatar: ";
$fieldset->add_child($legend);
$avatar = new HtmlElement("input", array("type"=>"file",
"name"=>"avatar", "id"=>"avatar"));
$label = new HtmlElement("label", array("for"=>"avatar"));
$label->innerHTML = "Avatar: ";
$desc = new HtmlElement("span", array("class" => "description",
"id" => "avatar_desc"));
$desc->innerHTML = "The file must be jpeg or gif with a size of
maximum 100 KB";
$div = new HtmlElement("div", array("class"=>"fieldline"));
```

```
$hidden = new HtmlElement("input", array("type"=>"hidden",
"name"=>"MAX_FILE_SIZE", "value"=>"300000"));
$div->add_child($hidden);
$div->add_child($label);
$div->add_child($avatar);
$div->add_child($desc);
$fieldset->add_child($div);
$submit = new HtmlElement("input",
array("type"=>"submit","value"=>"Change"));
$div = new HtmlElement("div", array("class"=>"submit fieldline"));
$div->add_child($submit);
$fieldset->add_child($div);
$form->add_child($fieldset);
```

After the developer runs the *code* method on the newly created element he gets the following HTML string:

```
<form enctype="multipart/form-data" class="edit_avatar"
method="post" action="cont/edit/avatar" >
  <fieldset>
    <legend>Change avatar: </legend>
    <div class="fieldline" >
      <input type="hidden" name="MAX_FILE_SIZE" value="300000" />
      <label for="avatar" >Avatar: </label>
      <input type="file" name="avatar" id="avatar" />
      <span class="description" id="avatar_desc" >
      The file must be jpeg or gif with a size of maximum 100 KB
      </span>
    </div>
    <div class="submit fieldline" >
      <input type="submit" value="Change" />
    </div>
  </fieldset>
</form>
```

Conclusion

Length of code might not be the biggest gain at first sight (although it will be after extending the class to create specialized Form and Table classes, as discussed in future articles), but DOM like manipulation is. The model also provides a way of taking HTML code out of scripting files for a more consistent look and feel of the code.

Acknowledgement

This work was co-financed from the European Social Fund through Sectorial Operational Program Human Resources Development 2007-2013, project number POSDRU/107/1.5/S/77213 „Ph.D. for a career in interdisciplinary economic research at the European standards”

Bibliography

1. *"Zend Framework in Action"*, Rob Allen, Nick Lo, Steven Brown; Manning Publications; 1 edition (January 4, 2009), ISBN-13: 978-1933988320
2. *"Zend Framework 1.8 Web Application Development"*, Keith Pope; Packt Publishing (October 26, 2009), ISBN-13: 978-1847194220
3. *"PHP Object-Oriented Solutions"*, David Powers; friendsofED; 1 edition (August 21, 2008), ISBN-13: 978-1430210115
4. *"Beginning ASP.NET 4: in C# and VB (Wrox Programmer to Programmer)"*, Imar Spaanjaars; Wrox; 1 edition (March 22, 2010), ISBN-13: 978-0470502211
5. *"Pro ASP.NET 4 in C# 2010"*, Matthew MacDonald, Adam Freeman; Apress; 4 edition (June 30, 2010), ISBN-13: 978-1430225294
6. *"Professional ASP.NET MVC 3 (Wrox Programmer to Programmer)"*, Jon Galloway, Phil Haack, Brad Wilson, K. Scott Allen; Wrox; 1 edition (August 9, 2011), ISBN-13: 978-1118076583