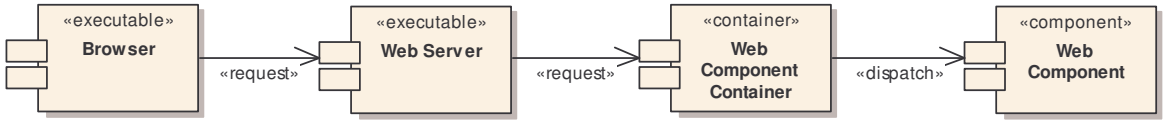
Dynamic Web Page

Allows data to by served dynamically to consumers. The implementation of this pattern may facilitate varying degrees of software development best-practises such as loose coupling and separation of concerns.



Background

A consumer requests information from a Web site in order to receive a view of pertinent live data. The business Web site must reply to the request with a well-formatted response that contains the data the consumer expects to see. The developers desire to use software development best practices in the development of the site. The business supports the developers' desires since it will promote reuse and maintainability, thereby lowering operation costs.

Value and Benefits

Dynamic Web pages are an invaluable business asset. They are matchless in their ability to deliver business information in a timely manner.

Putting It to Work

When a user request reaches the Web server it is in the form of a URL. The URL must contain some directives that inform the Web server how to handle the request. A basic request will have a URL that references a static Web page, graphic file, or other kind of file that can be served directly by the Web server. This is a basic URL:

```
http://www.somedomain.com/index.html
```

This URL requests a static Web page. Embedded references to graphics files are also in the form of URLs. However, there are no dynamic aspects to this request. If Web sites were limited to static content they would provide very limited value to enterprise businesses.

Custom Components

Web servers must, therefore, also handle requests for dynamic data. The Web server dispatches a URL containing a special directive to a special request handler. Note the difference between the two kinds of URLs:

```
http://www/somedomain.com/dyna-container/dyna-service?param1=x
```

The second URL has what could be considered a special directive. When correctly configured and the server receives a URL with the text pattern `/dyna-container/dyna-service`, the request is dispatched to the special request handler

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

mapped to the pattern in the server's configuration. In this example `dyna-container` represents some sort of technology request processing engine, and `dyna-service` is a custom component managed by the engine.

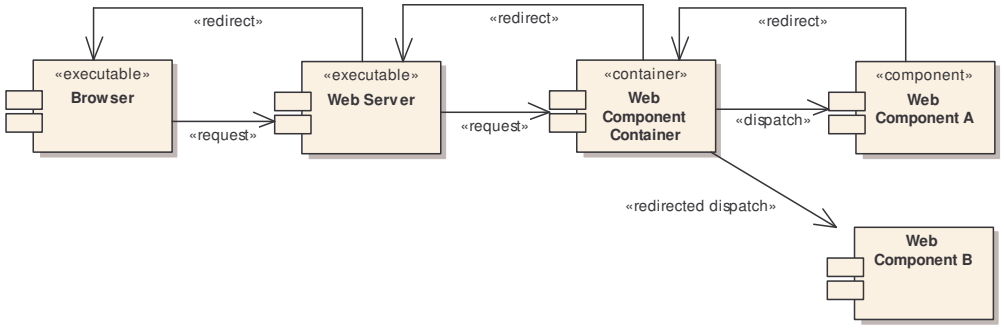
Because some request processing engines manage custom components privately, including their deployment and runtime execution, they may be called *containers*. Containers will many times provide other custom facilities, and resource access mechanisms, as well as a standard API for the components to use.

The request handler, or container, further parses the request URL and determines which of its custom components to dispatch the request to (`dyna-service`). It may also find parameters on the URL, such as `param1=x` above. The request handler must factor the parameters into a format or data structure suitable to be accepted as parameters to the specific component. Finally the request is dispatched to the custom component along with any parameters.

The custom component receives the request and performs some sort of processing. It may look up data in one or more data sources, or delegate such responsibilities to one or more other components. Whichever components are designed to produce all or some of the results in the response, they output information to an output stream. The output stream may be in memory or it may be written to a temporary file. In either case the response output must be appropriate for the Web server to use in its response to the requester. This usually means formatting the response in HTTP format. The inner response data is traditionally HTML, but may use other formats such as XML.

The process of delegating partial responsibility to other custom components takes two forms: components that are managed by the request processing engine or Web component container, and those managed outside (perhaps in another kind component container).

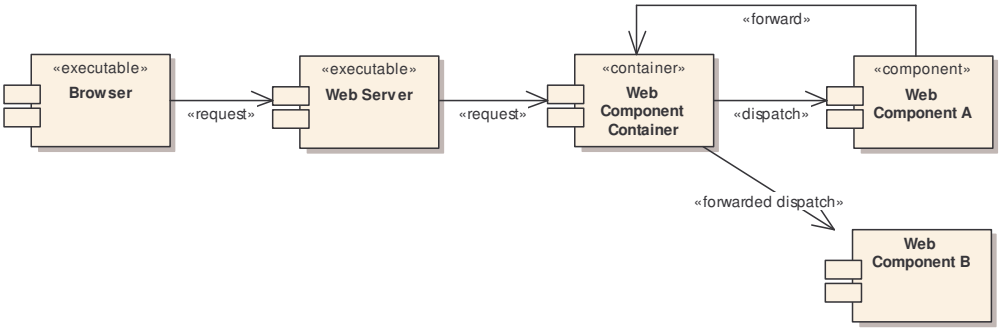
Components managed inside the Web component container may use an API to dispatch requests to other, sibling components. Dispatching takes the form of *redirecting* and *forwarding*. Redirecting changes the URL, and requires that the response go back to the browser (or other request sender) and then back to the Web server to be processed. Forwarding does not require that the request make a round-trip to the browser and back. Rather, it simply forces the request handling to be supplemented by the component that it forwards the request to. Note the difference between redirecting and forwarding as illustrated in the next two diagrams:



Web Component A redirects request handling to *Web Component B*. This requires the URL redirection to make a round-trip to the *Browser* and back to the *Web Component Container* before *Web Component B* receives the request. Essentially *Web Component A*

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

forces the *Browser* to make an entirely new request to the *Web Server* and tells the *Browser* what URL to request.



Here *Web Component A* forwards the request handling to *Web Component B*. The major difference here is that the *Web Component Contain* simply re-dispatches the request to *Web Component B* directly. No *Browser* round-trip is necessary. While this is a more optimal approach, the *Browser*'s address text box will contain the URL of the original request to *Web Component A*, rather than *Web Component B*. But an identical request to *Web Component A* will result in an identical response in the future; that is, as long as the component logic and backing data driving the request remain unchanged.

It is not within the scope of this pattern to define the use of components that reside outside the Web component container. But suffice it to say that this may be managed using the native API of the consumed component container (such as COM or EJB).

Template Pages and Scriptlets

In all of the above cases one or more custom component must generate the response output that gets served back to the requester. While that does fulfill the definition of a *Dynamic Web Page*, it is certainly not the most convenient to use. If you want to alter the presentation of the response, you must update the custom component, which is likely implemented in some sort of modern programming language. The output itself is difficult to produce because any HTML or XML output produced is being shoved through an output file stream. It's much more convenient to use an HTML or XML editor to create the page layout and then merge in runtime data on the fly as needed.

So another important aspect of the *Dynamic Web Page* is the use of *template pages* interspersed with programming logic hosted by a scripting syntax to produce the live data. The URL that requests a template page is generally different from those used to request custom components. In fact a template page is requested in the same way static HTML pages are requested:

```
http://www.somedomain.com/page.type
```

The filename suffix is important to the Web server's ability to dispatch the template page request. In the same way that a URL identifies a custom component, a URL also identifies a template page. The `.type` filename suffix tells they Web server to direct the request to the template page's special request handler.

Here is what the markup in a template page might look like:

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

```
<html>
<head>
<title>Acme, Inc.</title>
</head>
<body>
. . .
<table>
 <tr>
  <td>Info 1</td>
  <td><%= info1.getInfo() %></td>
 </tr>
 <tr>
  <td>Info 2</td>
  <td><%= info2.getInfo() %></td>
 </tr>
</table>

</body>
</html>
```

Without getting too deep into the meaning of the embedded scripting language (yet), note how concerns are divided. The HTML markup and standard presentation text is provided as natural, static HTML. Dynamic, live data, on the other hand, is brought into the page by executing the logic in the *scriptlets*. The scriptlet is the programming logic surrounded by the <%= and %> directives. So your page design, including graphics and layout, can be performed using an HTML editor (textual or WYSIWYG), and programming logic can be inserted later (or the other way around, depending on how your development is planned).

Depending on the implementation you use, the template pages may be generated as first-class custom components that are native to the Web component container, or they may be interpreted at runtime. If they are generated into native custom components, expect better performance than those that are interpreted.

A container that turns a template page into a native custom component will compare the date and time stamp of the template page with that of its corresponding custom component, if the generated component exists yet at all. If the template page is newer than the generated component, or if no corresponding generated component exists, then one is generated. From that point forward the generated component is executed in place of requests for the template page.

Custom Tag Libraries

A third important aspect of the *Dynamic Web Page* is the ability to extend or augment the tag-based markup language used by the template page. Basically the Web component container allows developers to define their own set of tags, housing them in a set of custom tag libraries. A class or other programming implementation mechanism backs the custom tags. The programming logic in the mechanism is executed at runtime when the page is requested and logical location of the tag is reached. The tags are used in place of most of the scriptlets, although scriptlets may be necessary to provide runtime information to the custom tags, as can be seen here:

```
<html>
<head>
<title>Acme, Inc.</title>
</head>
<body>
. . .
<table>
```


© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

```
 <tr>
  <td>Info 1</td>
  <td><info:show-info id="<%= info1.getId() %>"></td>
 </tr>
 <tr>
  <td>Info 2</td>
  <td><info:show-info id="<%= info2.getId() %>"></td>
 </tr>
</table>

</body>
</html>
```

The tags may provide the ability to use namespaces, such as can be seen above where `info` is the namespace and `show-info` is the name of the custom tag. The tag's `id` attribute is used to locate the actual information being displayed live. Here a scriptlet is used to get the identification of the specific "info" object to display. In essence the Web component container, when generating a representative native custom control, reads the custom tags and replaces the tag with an invocation of the programming logic provided by the backing implementation mechanism. By using a combination of custom components, template pages with scriptlets, and custom tag libraries, we have a good foundation on which to build out an enterprise-strength *Dynamic Web Site*.

Implementations and Examples

A somewhat antiquated means of supporting dynamic page generation is called *cgi-bin*. *CGI* stands for *Common Gateway Interface*, which represents a standard way to access live data via a Web request. The *bin* part of *cgi-bin* stands for *binary*, and is representative of the Unix environment. Recall that the `/bin` and `/usr/bin` directories, for example, are traditional locations where executable program files reside on a Unix system. So `cgi-bin` is the directory in which executable programs and utilities reside that allow a Web site to produce dynamic pages upon request.[1] When the Web URL contains the text `/cgi-bin/executable-file`, the Web server knows to run the specific `executable-file` that follows the `/cgi-bin/` directory reference.

There are many issues with using traditional *cgi-bin*. It is usually slow because an executable file must be loaded and run each time a *cgi-bin* request is made. Loading an executable file on most operating systems is one of the most expensive system-level services that can be employed. Also each request may require loading its own dedicated executable, putting even greater load on the system.

There are other issues, such as those concerning caching and security that I will not describe here (as they are generally well known). Because of the various issues around *cgi-bin*, its use today is rare. Other technologies such a *Java Servlets*, *JavaServer Pages* (JSP), *Active Server Pages* (ASP and ASP.NET), *PHP*, and others have largely replaced the use of *cgi-bin*.

The basic idea behind more modern solutions to the dynamic Web page problem domain is the use of a scripting language inside static pages. When the consumer requests a page containing a scriptlet, its logic is executed. The scriptlet logic produces dynamic

[1] Note that `cgi-bin` generally does not refer to a literal directory. More than likely the location is logical, an alias that tells the Web server that it needs to execute some runtime behavior. The actually location is usually unknown to the consumer in order to provide a measure of security.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

data that gets inserted into key areas of the output stream. The end result is that the page response contains a snapshot of live data at the time of the request. The scripting helps to separate programming logic from presentation. Here's an example using JSP:

```
<tr>
 <td>Wholesale Price:</td> <td><%= productDataChannel.getWholesalePrice() %></td>
</tr>
<tr>
 <td>Retail Price:</td> <td><%= productDataChannel.getRetailPrice() %></td>
</tr>
```

This is a portion of a JSP, which is basically an HTML document with embedded Java. This example shows a portion of an HTML table definition. The definition of two table rows is shown (surrounded by `<tr></tr>` tags), and each row has two columns (surrounded by `<td></td>` tags). The first row contains the wholesale price of a particular product, and the second row contains the same product's retail price. The first column of each row is the description text in plain HTML. The second column is filled with live data, using the following scriptlet notation:

```
<%= object.method () %>
```

This particular JSP syntax (`<%=expression %>`) directs that the `String` results of the expression should be inserted into the page at the given location. The `object` in both example expressions is `productDataChannel`. The `method` binding of the first expression is `getWholesalePrice()`, and that of the second expression is `getRetailPrice()`.

Well, this is very useful. But unless other advantages are realized the improvements over using *cgi-bin* may be somewhat marginal. For example, what about addressing performance concerns, data formatting, and design best practices?

Another advantage of using ASP.NET, JSP, and the like, is that pages are pre-compiled into native components, and therefore optimized (not so with original ASP). When the scripted page is requested the generated, pre-compiled native component is executed, which means that live data is accessed at the speed of the underlying implementation language. Depending on the selected technology, the speed is determined by compiled C# or VisualBasic, or by Java and its virtual machine, for example. In the case of JSP, the page definition is compiled into a *Java Servlet* class. This is accomplished by essentially inverting the JSP page. The scripting logic becomes the body of the servlet's `doService()` method, while the text of the HTML portion of the page becomes the core of the servlet's output to the response stream.

Many dynamic Web containers, such as a Java Servlet container, typically load only one instance of the scripted executable. This reduces memory and load-time overhead by reusing a single copy of the *Java Servlet*, which is actually an executable class object.

When ASP, JSP, and Java Servlets were first used, some problems were encountered because core business logic tended to creep into presentation logic (an even by design!). Efforts have been made to reduce this problem. Some of the solutions have simply been provide through education, such as with patterns. Other solutions have come in the form of technologies, such as tag libraries and frameworks.

Tag libraries allow software developers to create their own custom tags that can be inserted into pages. This is done much like inserting standard HTML tags. Using JSP tag

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

libraries, page developers must import the tag library into the page so it can be referenced. Next the tag must be referenced:

```
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
. . .
<netui:button value="Sign On" type="submit" style="font-family:verdana;font-size:7pt;"/>
```

The above example uses the custom tag library provided by *BEA* and its *WebLogic* platform. This example deals specifically with HTML syntax such as forms and their input fields. Here an HTML button is defined. The face of the button will have the text "Sign On," and if clicked will cause the form to be submitted to the Web site. Rather than using the scripting method to produce product-pricing information, as follows:

```
<tr>
 <td>Wholesale Price:</td> <td><%= productDataChannel.getWholesalePrice() %></td>
</tr>
```

What if you use a custom tag library instead?

```
<%@ taglib uri="channel-product-tags.tld" prefix="channel"%>
. . .
<tr>
 <td>
  <channel:product-wholesale-price-title/>
 </td>
 <td>
  <channel:product-wholesale-price
     style="font-family:verdana;font-size:7pt;"
     format="$999,999.99"
     product-id="<%= product.getId() %>"/>
 </td>
</tr>
```

Note in the above tag (`<channel:product-wholesale-price/>`) that display formatting is supported, whereas in the scriptlet implementation it was not considered. Of course formatting could have been supported in the scriptlet example, but it would have been more complex to accomplish. The strength of the custom tag approach is that you can easily manage both data access and presentation concerns in a page-centric manner, removing both from page layout concerns. The JSP custom tags are implemented using Java classes. Thus, the implementations of both access and formatting may be changed without impacting the pages themselves.

You also have the versatility to change the tag parameters. For example, instead of hard coding the style and format attributes, you may decide to use symbolic names:

```
<td>
  <channel:product-wholesale-price
     style="Std_Style"
     format="Locale_Currency"
     product-id="<%= product.getId() %>"/>
</td>
```

Now you simply leave it to dynamically executed presentation logic to map the actual display styles and formats used. In the previous example the `Std_Style` may be changed by the business whenever appropriate, or even map to a user preference. Further, the `Locale_Currency` parameter allows the business to present prices according to the locale of the consumer, making your *Dynamic Web Site* dynamically

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

internationalized. Presentation and business logic is not only separated, but a more powerful presentation engine can be facilitated more easily as well.

Consequences

You will find tradeoffs among the following competing forces within the *Dynamic Web Page* solution pattern.

- *Separation of Concerns and Programming Power*: An indispensable advantage of having a strong backing technology for Web development is the power it provides. When you can separate page design and layout from dynamic data access and formatting, your site will be more maintainable. When your selected technology allows you to embed dynamic data access into the page, you have the power to provide the kinds of solutions needed by consumers.
- *Natural Page Development:* The closer you can get to a fully tag-based page implementation to more responsibility that can be delegated to less technical developers. It will "feel" natural to developers who are already strong in HTML skills.
- *Ease of Use Versus Underlying Complexity*: The easier that you make development of pages, the more complexity that resides below the surface. Providing a set of custom tab libraries puts more power into the hands of less technical developers (see *Natural Page Development* above). However, designing a complete and highly reusable set of dynamically functioning tags will cost your technical development staff in terms of complexity and, therefore, effort.

Related Patterns

See the other solution patterns within the *Dynamic Web Site* EBP. Since most or all of the patterns in this catalog may be implemented as Web-based solutions, this pattern is a foundation to the entire catalog. In addition, the following patterns make use of the *Dynamic Web Page* solution pattern.

- *Business Portal (page #)*: This EBP makes use of the *Dynamic Web Page* solution pattern. Also the *Portlet Application (page #)* solution pattern makes heavy use of this solution pattern.

Frameworks and Tools

- **Frameworks**: The *Jakarta Struts* framework provides an implementation of all aspects of this pattern. The *Spring Framework* provides an alternative to Struts. Some believe that *Spring* is a more intuitive and easier to use Java-based Web framework than *Struts*.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.