Chapter 1

Dynamic Content Generation

1 Introduction

1.1 What is dynamic content?

This chapter will largely restrict itself to a simple and sensible definition similar to that of Raghavan et al[1]:

A page is said to be dynamic if some or all of its content is generated at run-time.

In other words, a page is dynamic if processing must be done to create it after the page-request has been received, and is static if the entire page already exists on the server prior to the receipt of the page-request.

This definition encompasses the most common and powerful facets of what is generically known as *dynamic content.* Some useful exceptions exist, which will be noted and discussed later in the chapter.

1.2 Why is dynamic content important?

In 1997, it was estimated that approximately 80% of the content on the web was dynamic, and that the ratio of dynamic to static content would continue to increase[2], suggesting strongly that today an even larger percentage of the web's content is dynamic. Indeed, one recent study suggests that the content available through searchable online databases all of which serve dynamic content by necessity is approximately 400 to 500 times that available in static web pages[3]. Clearly, then, dynamic content is so ubiquitous in the web as to be indispensible in any attempt to understand or fully utilize it.

Moreover, some types of functionality are very difficult to implement without dynamic page creation. Perhaps the simplest and most common example is of a page that must show data retrieved from some search, such as a query into an online database or a use of an online search engine such as Google. The query could be almost anything, meaning that statically storing all possible result pages is simply impossible - the amount of storage space required isn't even available. To see this, simply go to your favorite search engine and type in ten words chosen at random from a dictionary. Roughly speaking, then, each word is drawn approximately uniformly from a pool of, say, 10,000 words, meaning your particular search is only one from a space of $10,000^{10} = 10,000,000,000,000,000,000,000,000,000,000,000,000,000$ possible ten-word searches. If each molecule in a disk drive could store an entire search result web page, it would still take over six hundred million tons of hard drives to store that many pages. And yet, for any ten words, a search engine can make a result page; hence, dynamically creating pages potentially offers enormous savings over a brute-force static approach.

As a less trivial example, time-dependent data like a stock ticker also relies on dynamic creation of the result given to the user. With online stock trading becoming increasingly popular, good investment online investment tools are also becoming increasingly important. In an active market, out-of-date stock quotes can cost money, and so any online stock ticker that will be used for trading must have data that's as fresh and current as possible, calling for dynamic serving of the data.

As a final example, web pages are making increasing use of customization services in electronic commerce, including both portals such as MyYahoo and companies like Amazon.com. Customization is when the web pages served are tailored to each user based on information about that user, and will be covered in much more detail in a later chapter. When each user's experience of a web site is thus different, the only sensible way to generate the pages that create that experience is dynamically using the customization data of that user.

1.3 What are the types of dynamic content?

Three main types of dynamic content are generally recognized:

1. Client-Side Dynamism
2. Temporal Dynamism
3. Context Dynamism

1.3.1 Client-Side Dynamism

Client-side dynamic content, covers most of the exceptions to the general definition of dynamic content. The canonical example of client-side dynamism is a standard Java applet, which allows dynamic activity within the limits prescribed by the static Java code embedded in the web page; the page may appear to change greatly, but the changes are rooted in appearance, not data. With a strictly client-side dynamic application, whatever happens, no new data is being shipped over from the server, and only dynamic rearrangements of the static data already loaded (the page and the applet) are possible. This, while not fitting our definition of true dynamic content, can nevertheless be very appropriate for non-data-intensive uses, such as exposition (a tutorial applet, for example). More detail on the uses of client-side dynamism, along with the other types of dynamic content, can be found in the section on tools (section 3).

1.3.2 Temporal Dynamism

Temporal dynamism is simply when a page's content depends on the time at which it is accessed. The canonical example of this is the stock ticker, where the data to be displayed must be retrieved at access-time. A borderline case can be found in some older implementations of services such as weather report pages, where the data changes infrequently enough to allow data-driven updates of static pages rather than user-driven creation of a dynamic page. As this is essentially static page creation, this chapter will not deal with such approaches, although the relationship between this and caching of dynamic pages will be examined in the section on caching (section 5).

1.3.3 Context Dynamism

Finally, context dynamism is essentially the core of our definition of dynamic content. With this, the contents of the page depend on the context of the viewing, which covers a very wide range of possibilities. One of the most common sources of context is explicit information from the user, with the simplest example being search terms in a keyword query. Implicit information from the user is also a common source of context for uses such as customization - one example of implicit user information is that derived from cookies stored by the user's web browser.

Regardless of the source of the context, each possible value of the context potentially requires a unique page to be sent to the user, meaning that the dynamic content of the page is driven by this information about the circumstances surrounding each individual viewing of the location. In particular, the page served to the viewer must generally be generated anew with each viewing, in contrast to the most common usages of the other two types of dynamism. This on-demand creation is what makes dynamic content generation both powerful and difficult to handle.

Due to the vast range of the term *context* and the on-demand nature of that type of dynamism, the large majority of dynamically generated content falls into this last category. Accordingly, and due to the greater complexity of this type of dynamism, the discussion in this chapter will apply primarily to this type of dynamic content, with exceptions as noted (although temporal dynamism can on occasion be considered a subtype of context dynamism if the time of page access is considered implicit information about the user).

2 Issues to Consider

Often the need for dynamically generated content and the type of dynamism required are reasonably apparent. In general, any page which must display information requested (implicitly or explicitly) by the user must be dynamic and use context dynamism. Client-side dynamism can be used for pages with a fixed set of information to be given to the user, and temporal dynamism can be used for pages with a fixed set of information that changes at intervals that do not depend on users of that page.

Once decision on whether to make a page dynamic and which type of dynamism have been made, there are still important questions to consider regarding optimization of the process. A poorly-considered dynamic content generation setup can be enormously expensive in terms of time, processing requirements, and physical resources, meaning that significant thought should be devoted to discerning the best way to tackle each individual problem.

In particular, contrasting on-demand page creation with static web pages immediately suggests difficulties in applying the standard approaches to caching and serving, as well as potential concerns in processing requirements. Approaches to address each of these concerns exist, although with some tradeoffs at times. The details of each of these general issues will be examined in the particular cases of several common tools in the next section; the aim here is to lay an abstract foundation for consideration and to aid understanding the importance of each issue in the context of discussion of each tool.

2.1 Efficiency

Driving almost all of this consideration is the question of efficiency - how can we solve our problem and obtain sufficient responsiveness with a minimum of resources? To concretize the discussion, one estimate for the CPU time required to serve a static page is 2-10 CPU milliseconds, while dynamically generating a page often takes orders of magnitude more effort[4].

Serving dynamic pages, then, puts a far greater strain on the processor resources available to the server. Some of this work can be offloaded to the client instead, although this has tradeoffs involved that will be examined in the next section. Regardless, dynamically generated pages are more complex to create and send off than static ones - more than a simple file read is required. Indeed, for a query-result page, such as displaying a user's phone bill, a large database will have to be queried to obtain the data to create the page from.

One large issue is that, since the page is being generated based on the particular values of contextual data, some program logic must examine that controlling data and determine what sort of page is required based on it. With a heavily-used server, even handling each page by forking off a single process within a server program's execution can severely tax the processor, especially if nontrivial logic is involved. While attempts to address this will be examined in the section on tools, there is no real way to remove the need for concurrent processing of the data involved with many concurrent requests, necessitating much more processor power for a dynamic web server than one with static pages.

Further, each dynamically served page must draw its data from somewhere, and this data will often come from a store much larger than a single static page. With query results, for example, any particular page will generally show only a small fraction of the database. Unless the data source is trivial enough to store entirely in available memory, each page generation may require nontrivial disk accessing as the relevant data is determined and obtained. Since each disk access is not only likely to be more complex than for a static page access but also is much less likely to be fulfillable from memory due to the size of the data that may need to be examined, the server also suffers from greatly increased data access time.

2.2 Processing Location

The server is the obvious place to do all of the work of generating the page, since it is the location of the data being served; however, some of the processing can potentially be given to the client to do. Creating a page essentially requires sifting through some data store according to the requirements spelled out in the context of the page access and generating a page from what results in accordance with that context, and so either client or server can create the final page if the context, data, and processing logic are available.

Doing this processing on the server is the most general approach, and requires no special support in the client. In addition, there is no need to send the client more data than is required, since the client has no need to sift through it. This can be important if sensitive data is involved, and can be crucial if the datasets involved are large - the bandwidth to send the data to each client for local processing may simply be unavailable. Similarly, we've seen that server-local processing can put a crushing workload on the server if the rate of page request is high. One way to think of things is that if bandwidth and processing ability were both infinite, there would be no need to do processing anywhere except on the server; however, even near-infinite network and processing capacity are expensive.

Shunting some of the work to the client takes a portion of the load off of the server and distributes it among the clients, greatly easing a potential bottleneck. While the workload from too-frequent page accesses can cripple a server, it is generally quite manageable for a client to handle its own processing. Unfortunately, not all dynamic content can effectively transfer some of the work to the client; in particular, it's very common for it to be infeasible to send the datasets required for dynamic content generation to the client for processing for security, capability, or network reasons. Additionally, support for any offloading of work is required to be built into the client in some manner, such as the ability to interpret javascript.

In summation, the bulk of work involved in most context dynamism must be done by the server, although some applications can instead make effective use of client-side dynamism instead and greatly lessen the server workload. More detail on these client-side uses, along with examples, can be found under the tools section in the discussions of Javascript and Java applets, the most common tools for this.

2.3 Caching Dynamic Content

Since dynamically generating a page potentially requires large amounts of CPU time, relatively slow IO, and possibly additional network traffic, caching these pages promises to result in huge savings. Due to the on-demand nature of dynamic content, however, standard caching approaches often don't work very well; indeed, there is little point in caching a page which will only be viewed that one time anyway. Such truly dynamic pages should not be cached, and are generally marked as uncachable in their HTML.

For some types of dynamic content, however, large sites with heavy traffic may see many requests for identical or similar pages, particularly with semi-dynamic content (content that is not static but does not fit our definition of dynamic content). For example, temporal dynamism is particularly amenable to this. In an implementation for the official web site for the 1998 Olympic Games in Nagano, aggressive propagation of temporally dynamic pages to web caches allowed cache hit rates of almost 100% to be achieved, as opposed to approximately 80% for the official site for the 1996 Olympic Games done by the same group[4]. This caching made it possible for a single server to handle requests for several hundred dynamic pages per second.

For more general context dynamism, however, where a page would be generated based on information provided implicitly or explicitly by the user, such a scheme will not work. In such cases, a caching solution has not yet been found, and, indeed, may not be appropriate - many dynamic pages are customized to a user, and hence will tend to be accessed infrequently enough that caching would be essentially wasted. Research in dynamic caching is fairly new, so hopefully important developments will be forthcoming, perhaps following approaches such as partial-page caching.

2.4 Language

Not all technologies are equally expressive in terms of what sort of dynamism they will allow, and each language has its own areas of appropriateness. Javascript, for example, is excellent for applications in the restricted domain of client-side dynamism, while CGI scripts allow much more general dynamism at the cost of greater server resource usage.

Similarly, some languages are optimized for particular types of applications, either in terms of language power or ease of using that language for those applications, and a relatively simple language such as DHTML may at times be more appropriate to use within its range of usefulness than a more complex language like Java - it may be so much faster to do the programming that the theoretical superiority of the more general language is more of a hindrance than a help.

The appropriateness of a particular technology also comes into play when considering robustness, a big issue for web applications that often fail remotely instead of where the problem can easily be seen. Using an appropriate language for the job, such as Perl for text processing instead of a Java servelet, can help to increase the robustness of the application by simplification.

Finally, efficiency of each language for a task and in general is an important consideration when choosing what to use. Sometimes a language will be technically capable of doing everything that is required, but the processing needs are too great to be feasible (such as CGI being too slow in general to use in the application of [4]). Both efficiency of running and efficiency of maintenance are important, as with all applications.

2.5 Security

Common server hacks include entering invalid input strings to test the error-handling of the application, entering a hugely long input to attempt to overflow a buffer and crash or clobber an application, and simple denial-of-service attacks where the server is hit with huge numbers of spurious requests to increase its workload.

Common client hacks are usually attempts to read or write the local filesystem, open new network connections to the client, or denial-of-service attacks that involve large numbers of spurious drains on the CPU, disk, or network capabilities.

While some of these problems will be covered in more detail during the discussions of tools that suffer from them, it is more important to be aware of the existence of hacks such as these in general. For any critical web server, it is important to research the weaknesses of the tools chosen and the common attacks it may have to face, and then to design and implement with thwarting those in mind (such as by always handling invalid and overflow input explicitly). Information about what needs to be taken into account is too specific to each tool to include here, but can usually be found by contacting either user groups of that tool or security specialists.

3 What Are the Tools for Dynamic Content Generation?

3.1 Traditional Programming Languages Are Inappropriate

Dynamic content generation often requires the integration of different data sources such as databases, static file sets, or even entire websites. Because of this, traditional programming languages such as C, C++, or JAVA are often ill-suited for these tasks, largely due to the fact that they are geared towards creating data structures and algorithms from scratch. In the case of dynamic content generation, this is often unnecessary because such things are inherent in the data sources. For example, search tools available in a database obviate the need for creating them from scratch when programming dynamic content.

Instead, programming tools for dynamic content generation are geared towards integration and rapid adaptation.

The very notion of dynamic content implies change. Because of this, integration and rapid adaptation become desirable characteristics of dynamic content programming languages (subsequently referred to as scripting languages). Traditional programming languages fall short in these categories as they are compiled languages, thus slowing development time. Philip Greenspun of MIT recently observed:

"Even if you were some kind of circus freak programmer and were able to consistently write bug-free code, C would still be the wrong language because it has to be compiled. Making a small change in a Web page might involve dragging out the C compiler and then restarting the Web server program so that it would load the newly compiled version of your API extension."

Furthermore, strong typing forces traditional programs to be carefully planned in advanced and limits rapid future development. It should be no surprise that scripting languages are often interpreted languages with limited data types.

3.2 What Are the Available Scripting Languages?

The origin of today's scripting languages began with UNIX shells such as sh and csh. They allowed programmers to write scripts to automate common tasks. The success of UNIX shells eventually led programmer Larry Wall to incorporate many of their features as well as those of the C programming language into the scripting language Perl (Practical Extraction and Report Language) during the late 1980's. Perl would eventually become one of the most popular Internet scripting languages, largely due to its ease of use and the fact that it is open source.

The latter characteristic of Perl would lead to the development of today's most popular scripting language, PHP. Despite Perl's popularity, some web developers desired to embed scripting code directly into HTML markup. By doing so, they believed that quicker response times and improved security could be achieved. Early Perl hacks later evolved into PHP, which is a recursive acronym standing for PHP hypertext preprocessor. Currently, PHP is the premier scripting platform on the Internet. A recent Netcraft survey has shown that it is used on 5.1 million domains. Furthermore, it is found on nearly 35% of Apache Web Servers, the most commonly used web server on the Internet.

Although Perl and PHP derivatives are the predominant Internet scripting languages, another language called Tool Command Language (TCL) has attracted a large following. This is largely due to the fact that it shares many characteristics with Perl and PHP such as being open source and easy to use. While TCL has fewer users than Perl and PHP, it has attracted attention for being completely integrated with AOLServer, an open-source, enterprise-scale web server capable of handling 28,000 hits per second. In addition, a popular graphical user interface (GUI) building package called TK allows users to quickly and easily build GUIs by providing a TCL interface to the X-window system.

Despite the popularity of open-source Internet scripting languages, there are a wide variety of commercial packages available. Many businesses or individuals would rather pay for complete all-in-one scripting platforms rather than running their corporate networks on a collection of hacks and patches. While the latter may be effective and free, it may require the unique expertise of a small group of highly specialized programmers. One of the most ubiquitous commercial Internet scripting languages is Visual Basic from Microsoft. Straddling the line between full-fledged programming and scripting language, Visual Basic is used by nearly 3 million developers worldwide, insuring a large populace of knowledgeable programmers. However, a major limitation for developers is that Visual Basic is not cross-platform.

Another commercial product is Coldfusion from Allaire Corporation. While not as prevalent as a scripting platform such as Visual Basic, Coldfusion is currently used by over half the Fortune 500.

3.3 CGI (Common Gateway Interface)

CGI is not a scripting language, rather it is a standard web protocol for starting programs. CGI allows one to obtain data, process it accordingly, and then return it to the user.

In short, a CGI script would appear as follows:

1. User input is stored.
2. Process user input data.
3. Write the HTML response.

Step #2 can invoke any scripting language such as Perl, Tcl, etc. The benefit of writing a CGI script is that it is standardized. Thus, a CGI script can be migrated from Apache Web Server to AOLServer with ease. Despite the ubiquity of CGI, it is undesirable to write all of one's programs using CGI for dynamic content. This is due to the fact that CGI demands a large amount of processing overhead by starting a new process each time the CGI script is invoked. Thus, for web servers with scripting languages modifications, it is easier to simply to invoke programs directly using those mechanisms, circumventing the overhead generated by CGI. However, CGI remains a valuable tool for developers requiring small programs that may require migration. Deciding On a Scripting Language I: Existing Development Platforms and Web Servers

One could conceivably program all their dynamic content tools according to CGI standards, allowing them easy cross-platform migration and web server compatibility. However, the resulting processing overhead makes renders such pursuits impractical. Thus, there seem to be two primary considerations when choosing a scripting language. Namely, what is the developer's current development platform and their intended web server?

Developers are often looking to integrate their existing computing components or to place them on the web. Because of this, many developers are constrained in their choice of scripting languages. For example, a company running their machines on Solaris and using Oracle as a database wouldn't be able to choose Visual Basic as their scripting language.

Additionally, the developer's intended web server is another important consideration. For example, if one chooses to use a web server such as AOLserver because of it is free and highly scaleable, then TCL might be a logical choice as a scripting language since it is completely integrated with AOLserver and a wide variety of development tools and resources are available for TCL/AOLserver usage.

In short, the proper choice for a scripting language doesn't occur in a vacuum.

3.4 Deciding On a Scripting Language II: Choosing the Appropriate Tools for the Job

Scripting languages such as TCL or Perl are highly adaptable and can be used to handle a wide variety of programming tasks. Indeed, languages such as Visual Basic can be used to create full-fledged Windows applications. However, when choosing a programming language, it is pertinent not to only consider what the language is capable of, but the ease of use. In addition, while scripting languages are usually rapid adaptable, existing tools or resources make some scripting languages more extensible than others. Thus, there seem to be two major considerations when choosing a scripting language. These are:

1. Ease of use
2. Breadth of functionality

For businesses or large websites, there seem to be two additional concerns:

1. Enterprise Usage
2. Web Server compatibility

How do today's scripting languages compare in each of these categories? Here's a comparison chart of many of today's scripting languages:

```
Features

Tcl
PHP
Perl
Java
Script
Visual
Basic
Coldfusion

Ease of Use
```

```
Rapid Development
Rapid Evolution
Regular Expressions
Breadth of functionality
Easily extensible
Embeddable
Internet and Web-enabled
Cross platform
Enterprise usage
Internationalization support
Thread safe
Database access
Web server Compatibility
Microsoft Internet Information Server
Netscape Enterprise
AOLServer
Apache Web Server
```

3.5 Ease of Use

All of the above listed scripting languages share rapid development and evolution as characteristics. However, some scripting languages such as Visual Basic were not originally designed with only web usage in mind. As a result, some scripting languages handle certain data types such as regular expressions better than others. TCL is a particular good example of this, since the only available data type is a string – perfect for handling dynamic web content.

3.6 Breadth of Functionality

The functionality of scripting languages describes a wide variety of programming tasks. While all the scripting languages in this chapter are web and Internet enabled, some are highly extensible, while others allow code to be directly embedded into HTML. Here are some programming characteristics encapsulated by the notion of functionality:

1. Easily extensible
2. Embeddable

3.7 Easily Extensible

Scripting languages "out of the box" are highly functional, but programming dynamic content tools often requires highly customized software extensions. As a result, scripting languages with a large number of existing extensions are highly desirable. TCL stands out as a particular outstanding candidate with a wide variety of extensions available. Even during the early stages of TCL's existence, extensions were available for accessing databases such as Sybase (sybTCL) and

Oracle(oraTCL). Another extension, incr TCL, provides object oriented programming features in TCL. In addition, one of the most popular of TCL's extensions is the aforementioned TK toolkit, which facilities rapid development of GUIs.

Visual basic is another programming language that is highly extensible, largely due to its origin as a full-fledge programming language. Visual basic allows developers to integrate Microsoft's ActiveX controls, software components that allowed specialized functions to be added to websites. ActiveX was first supported in web browsers beginning with Microsoft's Internet Explorer 3.0. There are literally thousands of ActiveX controls available; some of them are free while others are products available from commercial vendors. Visual Basic's depth is highly desirable for developers with an existing Windows NT based development platform, since it allows rapid migration of existing Visual Basic applications to the web.

3.8 Embeddable

Embeddable scripting languages convey several advantages to developers. Among them are quicker response times, improved security and transparency to the end user. The importance of these characteristics can be see in the popularity of PHP, an embeddable version of Perl. The number of PHP mods installed on Apache Web Servers outnumbers of the number of Perl mods by nearly 3 to 1.

The power of embedding scripting languages is that it allows users to rapidly adapt them to their existing applications. A developer begins by assigning actions to a user interface that interacts with an existing application (such as a C program) via TCL or PHP. The interaction of the user to the application can be readily adapted by altering the TCL or PHP commands without recompiling the programming, which would be necessary if the developer had the user interface act directly with the application. Also, by embedding script languages into existing applications, developers are able to rapidly migrate existing applications into the web.

4 Tools

4.1 Server Side Includes

Server-Side Includes (SSI) allow you to add some common forms of dynamic content to your web pages. To use SSI, you write special commands inside HTML comments in your document. For example, say you want a web page to automatically display the current date whenever it is requested. You would add the following line to your HTML document:

```
Today is <!--#echo var="DATE_LOCAL" -->, thanks for asking.
```

This line specifies that you want to run the `echo` command, with the argument being the variable `DATE_LOCAL`. The client will see the following line when it receives the web page:

`Today is Wednesday, 04-Apr-2001 14:31:27 EDT, thanks for asking.`

How do server-side includes work? Figure 1.1 shows the sequence of events:

1. The client requests an SHTML (server-parsed HTML) file from the web server.
2. The web server retrieves the document from the filesystem, then searches it for comments containing the special commands. If any commands are encountered, they are interpreted by the server, and replaced with the appropriate text.
3. The dynamically-generated web page is returned to the client.

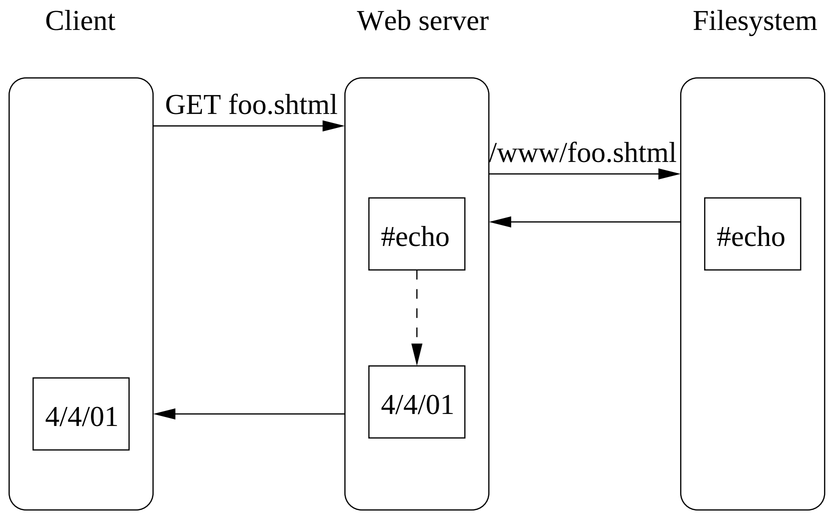

Figure 1.1: Server-side includes

SSI commands are mostly limited to very specific but useful tasks. Some common uses of SSI are:

- The `include` command inserts a specified file into that point in the HTML document. This command is useful for adding a common footer to each of your documents.
- The `config` command allows you to change the output format of other commands, such as the `echo` example above.
- The `echo` command can be used with the `LAST_MODIFIED` variable to automatically display the last modification date of the HTML document.

There is one extremely powerful command named `exec`. This command causes the web server to execute a CGI script or shell program, and insert the output of that program into the HTML document. However, this command is often disabled for security reasons.

SSI also supports one simple control-flow construct: the conditional expression. For example, you could output different greetings depending on which client has requested the web page:

```
<!--#if expr="$REMOTE_HOST = tolstoy.cs.brown.edu" -->
  Hello Dave.
<!--#else -->
  Hello stranger.
<!--#endif -->
```

The advantages of SSI are that they integrate directly with HTML source, and provide several useful operations for generating dynamic content. However, SSI have a couple drawbacks. For one, they don't give you the power of a full programming language. Also, they increase the load on the web server, since it has to parse each web page before sending it to the client.

Apache supports server-side includes via the `mod_include` module [ref].

4.2 Java Servlets

Java servlets give you the power to create dynamic content on the server side by implanting Java programs in the web server. The key feature of a servlet is that it keeps running across multiple requests. Other approaches we've seen, such as CGI, start up a new program for every access.

Let's first look at a simple servlet. You create a subclass of `HttpServlet` which overrides the `doGet()` method. This method is called every time the servlet is accessed by a web browser:

```
public class StanServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    {
        // set header information
        response.setContentType("text/html");

        // output document
        PrintWriter out = response.getWriter();

        out.println("<html><body>");
        out.println("<p>Hi Stan!</p>");
        out.println("</body></html>");

        out.close ();
    }
```

```
}
```

Sure, that was easy enough. But why are servlets any better than CGI or PHP? Remember, the servlet stays alive across multiple requests, and so does any data contained in the class. So, adding a counter to this web page is very easy; we just add a class variable and update it on each method call:

```
public class StanServlet extends HttpServlet
{
    private int counter;

    public void init ()
        throws ServletException
    {
        counter = 0;
    }

    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    {
        // set header information
        response.setContentType("text/html");

        // output document
        PrintWriter out = response.getWriter();

        out.println("<html><body>");
        out.println("<p>Hi Stan! Count is " + counter + ".</p>");
        out.println("</body></html>");

        out.close ();

        ++counter;
    }
}
```

To maintain state across requests with CGI, we would have to explicitly write the data to a file at the end of the program, then read it in again at the beginning of the program the next time it's invoked.

5 References

1. Raghavan et al: Crawling the Hidden Web
2. Lawrence and Giles: Searching the World Wide Web
3. The Deep Web: Surfacing Hidden Value.
   http://www.completeplanet.com/Tutorials/DeepWeb/.
4. Challenger et al: A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites