

Tutorials

mod_perl related tutorials, teaching you things not only about mod_perl, but also about any related topics of great interest to mod_perl programmers.

Last modified Thu Jan 29 08:40:38 2004 GMT

Part I: Application Design

- ▶ 1. Building a Large-Scale E-commerce site with Apache and mod_perl
mod_perl's speed and Perl's flexibility make them very attractive for large-scale sites. Through careful planning from the start, powerful application servers can be created for sites requiring excellent response times for dynamic content, such as EToys, all by using mod_perl.

Part II: Templating

- ▶ 2. Choosing a Templating System
Everything you wanted to know about templating systems and didn't dare to ask. Well, not everything....

Part III: Tips and Tricks

- ▶ 3. Cute Tricks With Perl and Apache
Perl and Apache play very well together, both for administration and coding. However, adding mod_perl to the mix creates a heaven for an administrator/programmer wanting to do cool things in no time!

Part IV: Client side facts and bugs

- ▶ 4. Workarounds for some known bugs in browsers.
Unfortunately for web programmers, browser bugs are not uncommon, and sometimes we have to deal with them; refer to this chapter for some known bugs and how you can work around them.
- ▶ 5. Web Content Compression FAQ

1 Building a Large-Scale E-commerce site with Apache and mod_perl

1.1 Description

mod_perl's speed and Perl's flexibility make them very attractive for large-scale sites. Through careful planning from the start, powerful application servers can be created for sites requiring excellent response times for dynamic content, such as EToys, all by using mod_perl.

This paper was first presented at ApacheCon 2001 in Santa Clara, California, and was later published by O'Reilly & Associates' Perl.com site: <http://perl.com/pub/a/2001/10/17/etoys.html>

1.2 Common Myths

When it comes to building a large e-commerce web site, everyone is full of advice. Developers will tell you that only a site built in C++ or Java (depending on which they prefer) can scale up to handle heavy traffic. Application server vendors will insist that you need a packaged all-in-one solution for the software. Hardware vendors will tell you that you need the top-of-the-line mega-machines to run a large site. This is a story about how we built a large e-commerce site using mainly open source software and commodity hardware. We did it, and you can do it too.

1.3 Perl Saves

Perl has long been the preferred language for developing CGI scripts. It combines supreme flexibility with rapid development. *Programming Perl* is still one of O'Reilly's top selling technical books, and community support abounds. Lately though, Perl has come under attack from certain quarters. Detractors claim that it's too slow for serious development work and that code written in Perl is too hard to maintain.

The mod_perl Apache module changes the whole performance picture for Perl. Embedding a Perl interpreter inside of Apache provides performance equivalent to Java servlets, and makes it an excellent choice for building large sites. Through the use of Perl's object-oriented features and some basic coding rules, you can build a set of code that is a pleasure to maintain, or at least no worse than other languages.

1.3.1 Roll Your Own Application Server

When you combine Apache, mod_perl, and open source code available from CPAN (the Comprehensive Perl Archive Network), you get a set of features equivalent to a commercial application server:

- Session handling
- Load balancing
- Persistent database connections
- Advanced HTML templating
- Security

You also get some things you won't get from a commercial product, like a direct line to the core development team through the appropriate mailing list, and the ability to fix problems yourself instead of waiting for a patch. Moreover, every part of the system is under your control, making you limited only by your team's abilities.

1.4 Case Study: eToys.com

When we first arrived at eToys in 1999, we found a situation that is probably familiar to many who have joined a growing startup Internet company. The system was based on CGI scripts talking to a MySQL database. Static file serving and dynamic content generation were sharing resources on the same machines. The CGI code was largely written in a Perl4-ish style and not as modular as it could be, which was not surprising since most of it was built as quickly as possible by a very small team.

Our major task was to figure out how to get this system to scale large enough to handle the expected Christmas traffic. The toy business is all about seasonality, and the difference between the peak selling season and the rest of the year is enormous. The site had barely survived the previous Christmas, and the MySQL database didn't look like it could scale much further.

The call had already been made to switch to Oracle, and a DBA team was in place. We didn't have enough time to do a re-design of the software, so we had to scramble to put in place whatever performance improvements we could finish by Christmas.

1.4.1 *Apache::PerlRun to the Rescue*

`Apache::PerlRun` is a module that exists to smooth the transition between basic CGI and `mod_perl`. It emulates a CGI environment, and provides some (but not all) of the performance benefits associated with code written for `mod_perl`. Using this module and the persistent database connections provided by `Apache::DBI`, we were able to do a basic port to `mod_perl` and Oracle in time for Christmas, and combined with some new hardware we were ready to face the Christmas rush.

The peak traffic lasted for eight weeks, most of which were spent frantically fixing things or nervously waiting for something else to break. Nevertheless, we made it through. During that time we collected the following statistics:

- 60 - 70,000 sessions/hour
- 800,000 page views/hour
- 7,000 orders/hour

According to Media Metrix, we were the third most heavily trafficked e-commerce site, right behind eBay and Amazon.

1.4.2 Planning the New Architecture

It was clear that we would need to do a re-design for 2000. We had reached the limits of the current system and needed to tackle some of the harder problems that we had been holding off on.

Goals for the new system included moving away from off-line page generation. The old system had been building HTML pages for every product and product category on the site in a batch job and dumping them out as static files. This was very effective when we had a small database of products since the static files gave such good performance, but we had recently added a children's bookstore to the site, which increased the size of our product database by an order of magnitude and made the time required to generate every page prohibitive. We needed a strategy that would only require us to build pages that customers were actually interested in and would still provide solid performance.

We also wanted to re-do the database schema for more flexibility, and structure the code in a more modular way that would make it easier for a team to share the development work without stepping on each other. We knew that the new codebase would have to be flexible enough to support a continuously evolving set of features.

Not all of the team had significant experience with object-oriented Perl, so we brought in Randal Schwartz and Damian Conway to do training sessions with us. We created a set of coding standards, drafted a design, and built our system.

1.5 Surviving Christmas 2000

Our capacity planning was for three times the traffic of the previous peak. That's what we tested to, and that's about what we got:

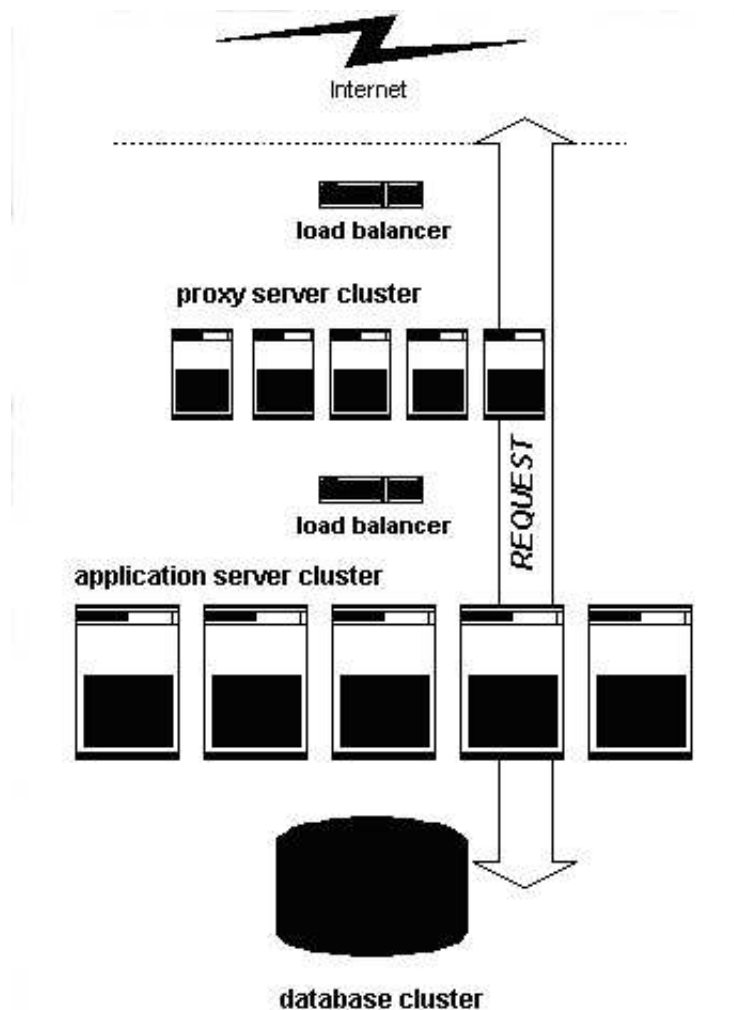
- 200,000+ sessions/hour
- 2.5 million+ page views/hour
- 20,000+ orders/hour

The software survived, although one of the routers went up in smoke. Once again, we were rated the third most highly trafficked e-commerce site for the season.

1.5.1 The Architecture

The machine strategy for the system is a fairly common one: low-cost Intel-based servers with a load-balancer in front of them, and big iron for the database.

Figure 1. Server layout



Like many commercial packages, we have separate systems for the front-end web servers (which we call proxy servers) and the application servers that generate the dynamic content. Both the proxy servers and the application servers are load-balanced using dedicated hardware from f5 Networks.

We chose to run Linux on our proxy and application servers, a common platform for mod_perl sites. The ease of remote administration under Linux made the clustered approach possible. Linux also provided solid security features and automated build capabilities to help with adding new servers.

The database servers were IBM NUMA-Q machines, which ran on the DYNIX/ptx operating system..

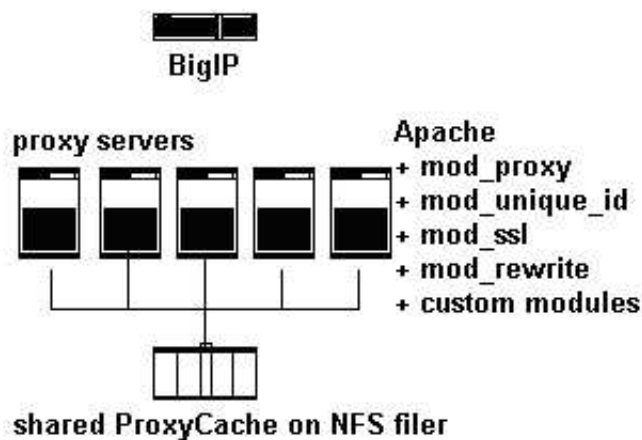
1.5.2 Proxy Servers

The proxy servers ran a slim build of Apache, without mod_perl. They have several standard Apache modules installed, in addition to our own customized version of mod_session, which assigned session cookies. Because the processes were so small, we could run up to 400 Apache children per machine. These servers handled all image requests themselves, and passed page requests on to the application servers. They communicated with the app servers using standard HTTP requests, and cached the page

results when appropriate headers are sent from the app servers. The cached pages were stored on a shared NFS partition of a Network Appliance filer. Serving pages from the cache was nearly as fast as serving static files.

This kind of reverse-proxy setup is a commonly recommended approach when working with `mod_perl`, since it uses the lightweight proxy processes to send out the content to clients (who may be on slow connections) and frees the resource-intensive `mod_perl` processes to move on to the next request. For more information on why this configuration is helpful, see the strategy section in the users guide.

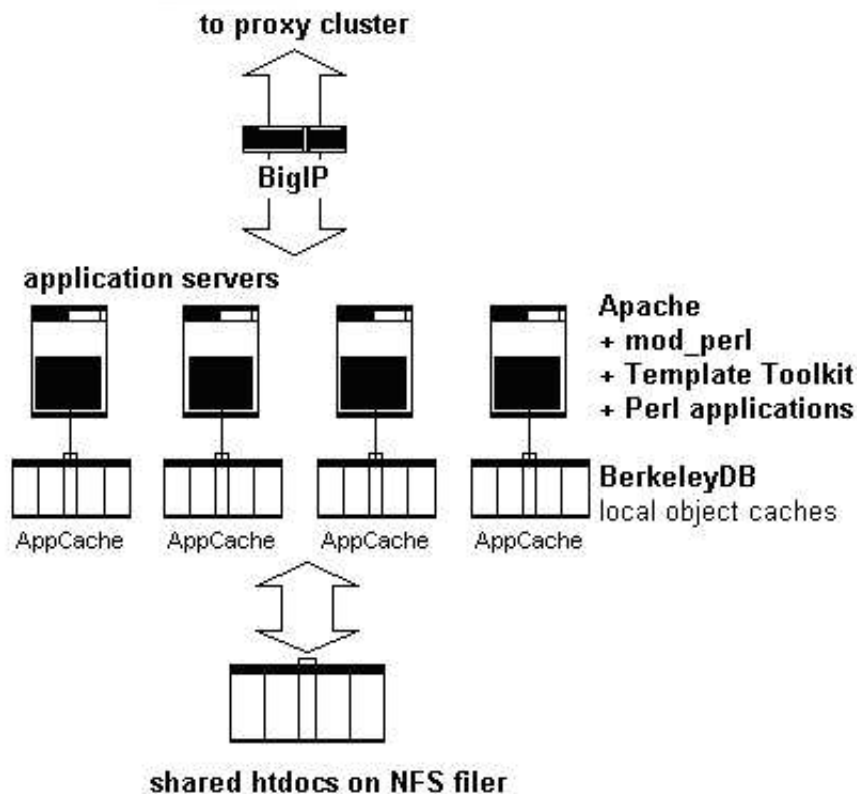
Figure 2. Proxy Server Setup



1.5.3 Application Servers

The application servers ran `mod_perl`, and very little else. They had a local cache for Perl objects, using Berkeley DB. The web applications ran there, and shared resources like HTML templates were mounted over NFS from the NetApp filer. Because they did the heavy lifting in this setup, these machines were somewhat beefy, with dual CPUs and 1GB of RAM each.

Figure 3. Application servers layout



1.5.4 Search servers

There was a third set of machines dedicated to handling searches. Since searching was such a large percentage of overall traffic, it was worthwhile to dedicate resources to it and take the load off the application servers and database.

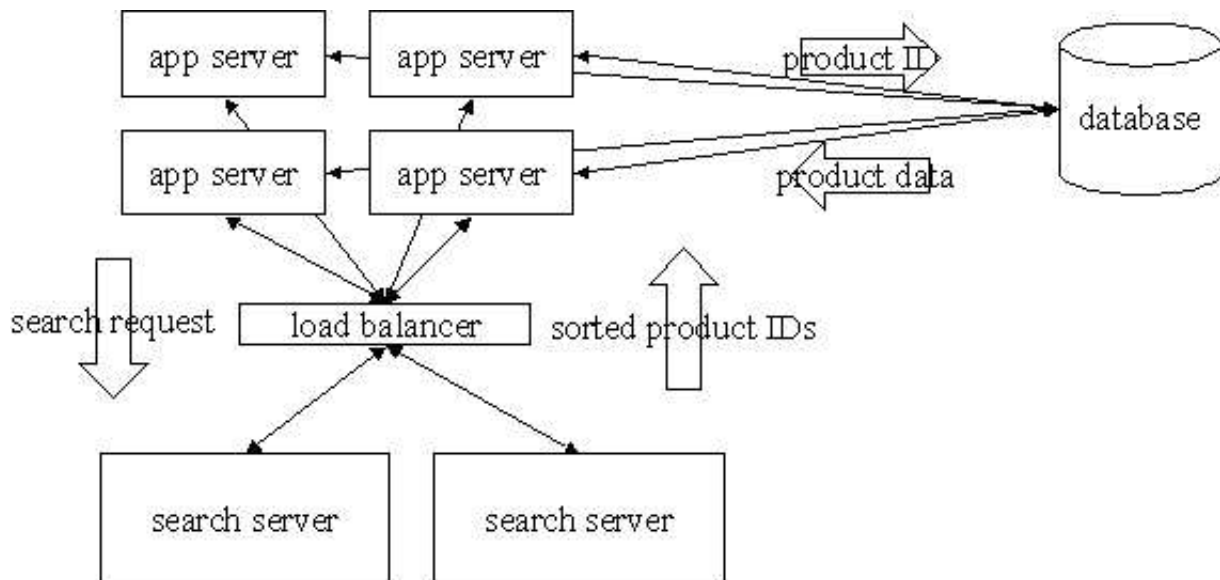
The software on these boxes was a multi-threaded daemon which we developed in-house using C++. The application servers talked to the search servers using a Perl module. The search daemon accepted a set of search conditions and returned a sorted list of object IDs of the products whose data fits those conditions. Then the application servers looked up the data to display these products from the database. The search servers knew nothing about HTML or the web interface.

This approach of finding the IDs with the search server and then retrieving the object data may sound like a performance hit, but in practice the object data usually came from the application server's cache rather than the database. This design allowed us to minimize the duplicated data between the database and the search servers, making it easier and faster to refresh the index. It also let us reuse the same Perl code for retrieving product objects from the database, regardless of how they were found.

The daemon used a standard inverted word list approach to searching. The index was periodically built from the relevant data in Oracle. There are modules on CPAN which implement this approach, including `Search::InvertedIndex` and `DBIx::FullTextSearch`. We chose to write our own because of the very tight performance requirements on this part of the system, and because we had an unusually

complex set of sorting rules for the returned IDs.

Figure 4. Search server layout



1.6 Load Balancing and Failover

We took pains to make sure that we would be able to provide load balancing among nodes of the cluster and fault tolerance in case one or more nodes failed. The proxy servers were balanced using a random selection algorithm. A user could end up on a different one on every request. These servers didn't hold any state information, so the goal was just to distribute the load evenly.

The application servers used “sticky” load balancing. That means that once a user went to a particular app server, all of her subsequent requests during that session were also passed to the same app server. The f5 hardware accomplished this using browser cookies. Using sticky load balancing on the app servers allowed us to do some local caching of user data.

The load balancers ran a periodic service check on every server and removed any servers that failed the check from rotation. When a server failed, all users that were “stuck”; to that machine were moved to another one.

In order to ensure that no data was lost if an app server died, all updates were written to the database. As a result, user data like the contents of a shopping cart was preserved even in cases of catastrophic hardware failure on an app server. This is essential for a large e-commerce site.

The database had a separate failover system, which we will not go into here. It followed standard practices recommended by our vendors.

1.7 Code Structure

The code was structured around the classic Model-View-Controller pattern, originally from SmallTalk and now often applied to web applications. The MVC pattern is a way of splitting an application's responsibilities into three distinct layers.

Classes in the Model layer represented business concepts and data, like products or users. These had an API but no end-user interface. They knew nothing about HTTP or HTML and could be used in non-web applications, like cron jobs. They talked to the database and other data sources, and managed their own persistence.

The Controller layer translated web requests into appropriate actions on the Model layer. It handled parsing parameters, checking input, fetching the appropriate Model objects, and calling methods on them. Then it determined the appropriate View to use and sent the resulting HTML to the user.

View objects were really HTML templates. The Controller passed data from the Model objects to them and they generated a web page. These were implemented with the Template Toolkit, a powerful templating system written in Perl. The templates had some basic conditional statements and looping in them, but only enough to express the formatting logic. No application control flow was embedded in the templates.

Figure 5. Code structure and interaction between the layers

1.8 Caching

The core of the performance strategy is a multi-tier caching system. On the application servers, data objects are cached in shared memory with a backing store on local disk. Applications specify how long a data object can be out of sync with the database, and all future accesses during that time are served from the high-speed cache. This type of cache control is known as "time-to-live." The local cache is implemented using a *Berkeley DB* database. Objects are serialized with the standard `Storable` module from CPAN.

Data objects are divided into pieces when necessary to provide finer granularity for expiration times. For example, product inventory is updated more frequently than other product data. By splitting the product data up, we can use a short expiration for inventory that keeps it in tighter sync with the database, while still using a longer expiration for the less volatile parts of the product data.

The application servers' object caches share product data between them using the IP Multicast protocol and custom daemons written in C. When a product is placed in the cache on one server, the data is replicated to the cache on all other servers. This technique is very successful because of the high locality of access in product data. During the 2000 Christmas season this cache achieved a 99% hit ratio, thus taking a large amount of work off the database.

In addition to caching the data objects, entire pages that are not user-specific, like product detail pages, can be cached. The application takes the shortest expiration time of the data objects used in the pages and specifies that to the proxy servers as a page expiration time, using standard *Expires* headers. The proxy servers cache the generated page on a shared NFS partition. Pages served from this cache have performance close to that of static pages.

To allow for emergency fixes, we added a hook to `mod_proxy` that deletes the cached copy of a specified URL. This was used when a page needed to be changed immediately to fix incorrect information.

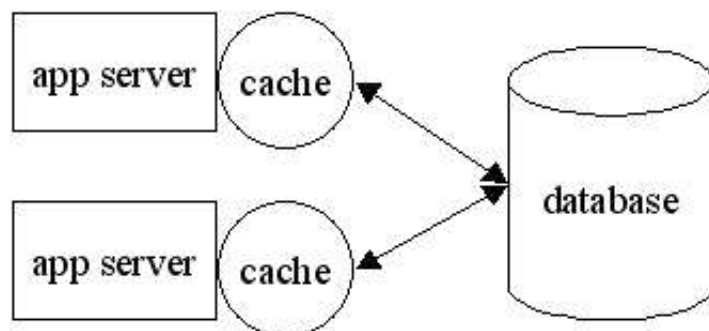
An extra advantage of this `mod_proxy` cache is the automatic handling of *If-Modified-Since* requests. We did not need to implement this ourselves since `mod_proxy` already provides it.

Figure 6. Proxy and Cache Interaction

1.9 Session Tracking

Users are assigned session IDs using HTTP cookies. This is done at the proxy servers by our customized version of `mod_session`. Doing it at the proxy ensures that users accessing cached pages will still get a session ID assigned. The session ID is simply a key into data stored on the server-side. User sessions are assigned to an application server and continue to use that server unless it becomes unavailable. This is called “sticky” load balancing. Session data and other data modified by the user -- such as shopping cart contents -- is written to both the object cache and the database. The double write carries a slight performance penalty, but it allows for fast read access on subsequent requests without going back to the database. If a server failure causes a user to be moved to a different application server, the data is simply fetched from the database again.

Figure 7. Session tracking and caches



1.10 Security

A large e-commerce site is a popular target for all types of attacks. When designing such a system, you have to assume that you will be attacked and build with security in mind, at the application level as well as the machine level.

The main rule of thumb is “don’t trust the client!” User-specific data sent to the client is protected using multiple levels of encryption. SSL keeps sensitive data exchanges private from anyone snooping on network traffic. To prevent “session hijacking” (when someone tampers with their session ID in order to gain access to another user’s session), we include a Message Authentication Code (MAC) as part of the session cookie. This is generated using the standard `Digest::SHA1` module from CPAN, with a seed phrase known only to our servers. By running the ID from the session cookie through this MAC algorithm we can verify that the data being presented was generated by us and not tampered with.

In situations where we need to include some state information in an HTML form or URL and don't want it to be obvious to the user, we use the CPAN `Crypt::` modules to encrypt and decrypt it. The `Crypt::CBC` module is a good place to start.

To protect against simple overload attacks, when someone uses a program to send high volumes of requests at our servers hoping to make them unavailable to customers, access to the application servers is controlled by a throttling program. The code is based on some work by Randal Schwartz in his `Stonehenge::Throttle` module. Accesses for each user are tracked in compact logs written to an NFS partition. The program enforces limits on how many requests a user can make within a certain period of time.

For more information on web security concerns including the use of MAC, encryption, and overload prevention, we recommend looking at the books *CGI Programming with Perl, 2nd Edition* and *Writing Apache Modules with Perl and C*, both from O'Reilly.

1.11 Exception Handling

When planning this system, we considered using Java as the implementation language. We decided to go with Perl, but we really missed Java's nice exception handling features. Luckily, Graham Barr's `Error` module from CPAN supplies similar capabilities in Perl.

Perl already has support for trapping runtime errors and passing exception objects, but the `Error` module adds some nice syntactic sugar. The following code sample is typical of how we used the module:

```
try {
    do_some_stuff();
} catch My::Exception with {
    my $E = shift;
    handle_exception($E);
};
```

The module allows you to create your own exception classes and trap for specific types of exceptions.

One nice benefit of this is the way it works with DBI. If you turn on DBI's *RaiseError* flag and use `try` blocks in places where you want to trap exceptions, the `Error` module can turn DBI errors into simple `Error` objects.

```
try {
    $sth->execute();
} catch Error with {
    # roll back and recover
    $dbh->rollback();
    # etc.
};
```

This code shows a condition where an error would indicate that we should roll back a database transaction. In practice, most DBI errors indicate something unexpected happened with the database and the current action can't continue. Those exceptions are allowed to propagate up to a top-level `try{}` block that encloses the whole request. When errors are caught there, we log a stacktrace and send a friendly error page back to the user.

1.12 Templates

Both the HTML and the formatting logic for merging application data into it is stored in the templates. They use a CPAN module called *Template Toolkit*, which provides a simple but powerful syntax for accessing the Perl data structures passed to them by the application. In addition to basics like looping and conditional statements, it provides extensive support for modularization, allowing the use of includes and macros to simplify template maintenance and avoid redundancy.

We found *Template Toolkit* to be an invaluable tool on this project. Our HTML coders picked it up very quickly and were able to do nearly all of the templating work without help from the Perl coders. We supplied them with documentation of what data would be passed to each template and they did the rest. If you have never experienced the joy of telling a project manager that the HTML team can handle his requested changes without any help from you, you are seriously missing out!

Template Toolkit compiles templates into Perl bytecode and caches them in memory to improve efficiency. When template files change on disk they are picked up and re-compiled. This is similar to how other `mod_perl` systems like Mason and `Apache::Registry` work.

By varying the template search path, we made it possible to assign templates to particular sections of the site, allowing a customized look and feel for specific areas. For example, the page header template in the bookstore section of the site can be different from the one in the video game store section. It is even possible to serve the same data with a different appearance in different parts of the site, allowing for co-branding of content.

This is a sample of what a basic loop looks like when coded in *Template Toolkit*:

```
[% FOREACH item = cart.items %]
name: [% item.name %]
price: [% item.price %]
[% END %]
```

1.13 Controller Example

Let's walk through a simple Hello World example that illustrates how the Model-View-Controller pattern is used in our code. We'll start with the controller code.

```
package ESF::Control::Hello;
use strict;
use ESF::Control;
@ESF::Control::Hello::ISA = qw(ESF::Control);
use ESF::Util;
sub handler {
    ### do some setup work
    my $class = shift;
    my $apr = ESF::Util->get_request();

    ### instantiate the model
    my $name = $apr->param('name');

    # we create a new Model::Hello object.
```

```

my $hello = ESF::Model::Hello->new(NAME => $name);

### send out the view
my $view_data{'hello'} = $hello->view();

# the process_template() method is inherited
# from the ESF::Control base class
$class->process_template(
    TEMPLATE => 'hello.html',
    DATA    => \%view_data);
}

```

In addition to the things you see here, there are a few interesting details about the `ESF::Control` base class. All requests are dispatched to the `ESF::Control->run()` method first, which wraps them in a `try{}` block before calling the appropriate `handler()` method. It also provides the `process_template()` method, which runs Template Toolkit and then sends out the results with appropriate HTTP headers. If the Controller specifies it, the headers can include Last-Modified and Expires, for control of page caching by the proxy servers.

Now let's look at the corresponding Model code.

```

package ESF::Model::Hello;
use strict;
sub new {
    my $class = shift;
    my %args = @_;
    my $self = bless {}, $class;
    $self{'name'} = $args{'NAME'} || 'World';
    return $self;
}

sub view {
    # the object itself will work for the view
    return shift;
}

```

This is a very simple Model object. Most Model objects would have some database and cache interaction. They would include a `load()` method which accepts an ID and loads the appropriate object state from the database. Model objects that can be modified by the application would also include a `save()` method.

Note that because of Perl's flexible OO style, it is not necessary to call `new()` when loading an object from the database. The `load()` and `new()` methods can both be constructors for use in different circumstances, both returning a blessed reference.

The `load()` method typically handles cache management as well as database access. Here's some pseudo-code showing a typical `load()` method:

```

sub load {
    my $class = shift;
    my %args = @_;
    my $id = $args{'ID'};
    my $self;
    unless ($self = _fetch_from_cache($id)) {
        $self = _fetch_from_database($id);
        $self->_store_in_cache();
    }
    return $self;
}

```

The save method would use the same approach in reverse, saving first to the cache and then to the database.

One final thing to notice about our Model class is the `view()` method. This method exists to give the object an opportunity to shuffle its data around or create a separate data structure that is easier for use with a template. This can be used to hide a complex implementation from the template coders. For example, remember the partitioning of the product inventory data that we did to allow for separate cache expiration times? The product Model object is really a façade for several underlying implementation objects, but the `view()` method on that class consolidates the data for use by the templates.

To finish off our Hello World example, we need a template to render the view. This one will do the job:

```

<html>
<title>Hello, My Oyster</title>
<body>
    [% PROCESS header.html %]
    Hello [% hello.name %]!
    [% PROCESS footer.html %]
</body>
</html>

```

1.14 Performance Tuning

Since Perl code executes so quickly under `mod_perl`, the performance bottleneck is usually at the database. We applied all the documented tricks for improving `DBD::Oracle` performance. We used bind variables, `prepare_cached()`, `Apache::DBI`, and adjustments to the `RowCache` buffer size.

The big win of course is avoiding going to the database in the first place. The caching work we did had a huge impact on performance. Fetching product data from the *Berkeley DB* cache was about ten times faster than fetching it from the database. Serving a product page from the proxy cache was about ten times faster than generating it on the application server from cached data. Clearly the site would never have survived under heavy load without the caching.

Partitioning the data objects was also a big win. We identified several different subsets of product data that could be loaded and cached independently. When an application needed product data, it could specify which subset was required and skip loading the unnecessary data from the database.

Another standard performance technique we followed was avoiding unnecessary object creation. The `Template` object is created the first time it's used and then cached for the life of the Apache process. Socket connections to search servers are cached in a way similar to what `Apache::DBI` does for database connections. Resources that are used frequently within the scope of a request, such as database handles and session objects, were cached in `mod_perl`'s `$r->pnotes()` until the end of the request.

1.15 Trap: Nested Exceptions

When trying out a new technology like the `Error` module, there are bound to be some things to watch out for. We found a certain code structure that causes a memory leak every time it is executed. It involves nested `try{}` blocks, and looks like this:

```
my $foo;
try {
    # some stuff...
    try {
        $foo++;
        # more stuff...
    } catch Error with {
        # handle error
    };
} catch Error with {
    # handle other error
};
```

It's not Graham Barr's fault that this leaks; it is simply a by-product of the fact that the `try` and `catch` keywords are implemented using anonymous subroutines. This code is equivalent to the following:

```
my $foo;
$subref1 = sub {
    $subref2 = sub {
        $foo++;
    };
};
```

This nested subroutine creates a closure for `$foo` and will make a new copy of the variable every time it is executed. The situation is easy to avoid once you know to watch out for it.

1.16 Berkeley DB

One of the big wins in our architecture was the use of *Berkeley DB*. Since most people are not familiar with its more advanced features, we'll give a brief overview here.

The `DB_File` module is part of the standard Perl distribution. However, it only supports the interface of *Berkeley DB* version 1.85, and doesn't include the interesting features of later releases. To get those, you'll need the `BerkeleyDB.pm` module, available from CPAN. This module can be tricky to build, but comprehensive instructions are included.

Newer versions of *Berkeley DB* offer many features that help performance in a `mod_perl` environment. To begin with, database files can be opened once at the start of the program and kept open, rather than opened and closed on every request. *Berkeley DB* will use a shared memory buffer to improve data access speed for all processes using the database. Concurrent access is directly supported with locking handled for you by the database. This is a huge win over `DB_File`, which requires you to do your own locking. Locks can be at a database level, or at a memory page level to allow multiple simultaneous writers. Transactions with rollback capability are also supported.

This all sounds too good to be true, but there are some downsides. The documentation is somewhat sparse, and you will probably need to refer to the C API if you need to understand how to do anything complicated.

A more serious problem is database corruption. When an Apache process using *Berkeley DB* dies from a hard kill or a segfault, it can corrupt the database. A corrupted database will sometimes cause subsequent opening attempts to hang. According to the people we talked to at Sleepycat Software (which provides commercial support for *Berkeley DB*), this can happen even with the transactional mode of operation. They are working on a way to fix the problem. In our case, none of the data stored in the cache was essential for operation so we were able to simply clear it out when restarting an application server.

Another thing to watch out for is deadlocks. If you use the page-level locking option, you have to handle deadlocks. There is a daemon included in the distribution that will watch for deadlocks and fix them, or you can handle them yourself using the C API.

After trying a few different things, we recommend that you use database-level locking. It's much simpler, and cured our problems. We didn't see any significant performance hit from switching to this mode of locking. The one thing you need to watch out for when using exclusive database level write locks are long operations with cursors that tie up the database. We split up some of our operations into multiple writes in order to avoid this problem.

If you have a good C coder on your team, you may want to try the alternate approach that we finally ended up with. You can write your own daemon around *Berkeley DB* and use it in a client/server style over Unix sockets. This allows you to catch signals and ensure a safe shutdown. You can also write your own deadlock handling code this way.

1.17 Valuable Tools

If you plan to do any serious Perl development, you should really take the time to become familiar with some of the available development tools. The debugger in particular is a lifesaver, and it works with `mod_perl`. There is a profiler called `Devel::DProf`, which also works with `mod_perl`. It's definitely the place to start when performance tuning your application.

We found the ability to run our complete system on individual's workstations to be extremely useful. Everyone could develop on his own machine, and coordinate changes using *CVS* source control.

For object modeling and design, we used the open source *Dia* program and *Rational Rose*. Both support working with UML and are great for generating pretty class diagrams for your cubicle walls.

1.18 Do Try This at Home

Since we started this project, a number of development frameworks that offer support for this kind of architecture have come out. We don't have direct experience using these, but they have a similar design and may prove useful to you if you want to take an MVC approach with your system.

`Apache::PageKit` is a `mod_perl` module available from CPAN which provides a basic MVC structure for web applications. It uses the `HTML::Template` module for building views.

OpenInteract is a recently released web application framework in Perl, which works together with the persistence layer *SPOPS*. Both are available from CPAN.

The *Application Toolkit* from Extropia is a comprehensive set of Perl classes for building web apps. It has excellent documentation and takes good advantage of existing CPAN modules. You can find it on <http://www.extropia.com/>.

If you want a ready-to-use cache module, take a look at the Perl-cache project on <http://sourceforge.net/>. This is the next generation of the popular `File::Cache` module.

The Java world has many options as well. The *Struts* framework, part of the *Jakarta* project, is a good open source choice. There are also commercial products from several vendors that follow this sort of design. Top contenders include *ATG Dynamo*, *BEA WebLogic*, and *IBM WebSphere*.

1.19 An Open Source Success Story

By building on the open source software and community, we were able to create a top-tier web site with a minimum of cost and effort. The system we ended up with is scalable to huge amounts of traffic. It runs on mostly commodity hardware making it easy to grow when the need arises. Perhaps best of all, it provided tremendous learning opportunities for our developers, and made us a part of the larger development community.

We've contributed patches from our work back to various open source projects, and provided help on mailing lists. We'd like to take this opportunity to officially thank the open source developers who contributed to projects mentioned here. Without them, this would not have been possible. We also have to thank the hardworking web developers at eToys. The store may be closed, but the talent that built it lives on.

1.20 Maintainers

The maintainer is the person(s) you should contact with updates, corrections and patches.

Per Einar Ellefsen <per.einar (at) skynet.be>

1.21 Authors

- **Bill Hilf** <bill (at) hilfworks.com>
- **Perrin Harkins** <perrin (at) elem.com>

Only the major authors are listed above. For contributors see the Changes file.

2 Choosing a Templating System

2.1 Description

Everything you wanted to know about templating systems and didn't dare to ask. Well, not everything....

2.2 Introduction

Go on, admit it: you've written a templating system. It's okay, nearly everyone has at some point. You start out with something beautifully simple like `$HTML =~ s/\$(\w+)/${$1}/g` and end up adding conditionals and loops and includes until you've created your very own unmaintainable monster.

Luckily for you, you are not the first to think it might be nice to get the HTML out of your code. Many have come before, and more than a few have put their contributions up on CPAN. At this time, there are so many templating modules on CPAN that it's almost certain you can find one that meets your needs. This document aims to be your guide to those modules, leading you down the path to the templating system of your dreams.

And, if you just went straight to CPAN in the first place and never bothered to write your own, congratulations: you're one step ahead of the rest of us.

2.2.1 *On A Personal Note*

Nothing can start an argument faster on the `mod_perl` mailing list than a claim that one approach to templating is better than another. People get very attached to the tools they've chosen. Therefore, let me say up front that I am biased. I've been at this for a while and I have opinions about what works best. I've tried to present a balanced appraisal of the features of various systems in this document, but it probably won't take you long to figure out what I like. Besides, attempts to be completely unbiased lead to useless documents that don't contain any real information. So take it all with a pound of salt and if you think I've been unfair to a particular tool through a factual error or omission, let me know.

2.3 Why Use Templates?

Why bother using templates at all? `Print statements` and `CGI.pm` were good enough for Grandpa, so why should you bother learning a new way to do things?

2.3.1 *Consistency of Appearance*

It doesn't take a genius to see that making one navigation bar template and using it in all of your pages is easier to manage than hard-coding it every where. If you build your whole site like this, it's much easier to make site-wide changes in the look and feel.

2.3.2 Reusability

Along the same lines, building a set of commonly used components makes it easier to create new pages.

2.3.3 Better Isolation from Changes

Which one changes more often, the logic of your application or the HTML used to display it? It actually doesn't matter which you answered, as long as it's one of them. Templates can be a great abstraction layer between the application logic and the display logic, allowing one to be updated without touching the other.

2.3.4 Division of Labor

Separating your Perl code from your HTML means that when your marketing department decides everything should be green instead of blue, you don't have to lift a finger. Just send them to the HTML coder down the hall. It's a beautiful thing, getting out of the HTML business.

Even if the same people in your organization write the Perl code and the HTML, you at last have the opportunity for more people to be working on the project in parallel.

2.4 What Are the Differences?

Before we look at the available options, let's go through an explanation of some of the things that make them different.

2.4.1 Execution Models

Although some try to be flexible about it, most templating systems expect you to use some variation of the two basic execution models, which I will refer to as "pipeline" and "callback." In the callback style, you let the template take over and it has the application's control flow coded into it. It uses callbacks to modules or snippets of in-line Perl code to retrieve data for display or perform actions like user authentication. Some popular examples of systems using this model include Mason, Embperl, and Apache::ASP.

The pipeline style does all the work up front in a standard CGI or mod_perl handler, then decides which template to run and passes some data to it. The template has no control flow logic in it, just presentation logic, e.g. show this graphic if this item is on sale. Popular systems supporting this approach include HTML::Template and Template Toolkit.

The callback model works very well for publishing-oriented sites where the pages are essentially mix and match sets of articles and lists. Ideally, a site can be broken down into visual "components" or pieces of pages which are general enough for an HTML coder to re-combine them into entirely new kinds of pages without any help from a programmer.

The callback model can get a bit hairy when you have to code logic that can result in totally different content being returned. For example, if you have a system that processes some form input and takes the user to different pages depending on the data submitted. In these situations, it's easy to end up coding a spaghetti of includes and redirects, or putting what are really multiple pages in the same file.

On the other hand, a callback approach can result in fewer files (if the Perl code is in the HTML file), and feels easier and more intuitive to many developers. It's a simple step from static files to static files with a few in-line snippets of code in them. This is part of why PHP is so popular with new developers.

The pipeline model is more like a traditional model-view-controller design. Working this way can provide additional performance tuning opportunities over an approach where you don't know what data will be needed at the beginning of the request. You can aggregate database queries, make smarter choices about caching, etc. It can also promote a cleaner separation of application logic and presentation. However, this approach takes longer to get started with since it's a bigger conceptual hurdle and always involves at least two files: one for the Perl code and one for the template.

Keep in mind, many systems offer significant flexibility for customizing their execution models. For example, Mason users could write separate components for application logic and display, letting the logic components choose which display component to run after fetching their data. This allows it to be used in a pipeline style. A Template Toolkit application could be written to use a simple generic handler (like the `Apache::Template` module included in the distribution) with all the application logic placed in the template using object calls or in-line Perl. This would be using it in a callback style.

`HTML::Template` and some of the `AxKit` XML processors are fairly rigid about insisting on a pipeline approach. Neither provide methods for calling back into Perl code during the HTML formatting stage; you have to do the work before running the template. The authors of these tools consider this a feature since it prevents developers from cheating on the separation of application code and presentation.

2.4.2 Languages

Here's the big issue with templating systems. This is the one that always cranks up the flame on web development mailing lists.

Some systems use in-line Perl statements. They may provide some extra semantics, like `Embperl`'s operators for specifying whether the code's output should be displayed or `Mason`'s `<%init>` sections for specifying when the code gets run, but at the end of the day your templates are written in Perl.

Other systems provide a specialized mini-language instead of (or in addition to) in-line Perl. These will typically have just enough syntax to handle variable substitution, conditionals, and looping. `HTML::Template` and `Template Toolkit` are popular systems using this approach. `AxKit` straddles the fence, providing both a (not-so-) mini-language - `XSLT` - and an in-line Perl approach - `XPathScript`.

Here's how a typical discussion of the merits of these approaches might go:

IN-LINE: Mini-languages are stupid. I already know Perl and it's easy enough. Why would you want to use something different?

MINI-LANG: Because my HTML coder doesn't know Perl, and this is easier for him.

IN-LINE: Maybe he should learn some Perl. He'd get paid more.

MINI-LANG: Whatever. You just want to use in-line Perl so you can handle change requests by putting little hacks in the template instead of changing your modules. That's sloppy coding.

IN-LINE: That's efficient coding. I can knock out data editing screens in half the time it takes you, and then I can go back through, putting all the in-line code into modules and just have the templates call them.

MINI-LANG: You could, but you won't.

IN-LINE: Is it chilly up there in that ivory tower?

MINI-LANG: Go write some VBScript, weenie.

etc.

Most people pick a side in this war and stay there. If you are one of the few who hasn't fully decided yet, you should take a moment to think about who will be building and maintaining your templates, what skills those people have, and what will allow them to work most efficiently.

Here's an example of a simple chunk of template using first an in-line style (Apache::ASP in this case) and then a mini-language style (Template Toolkit). This code fetches an object and displays some properties of it. The data structures used are identical in both examples. First Apache::ASP:

```
<% my $product = Product->load('sku' => 'bar1234'); %>

<% if ($product->isbn) { %>
    It's a book!
<% } else { %>
    It's NOT a book!
<% } %>

<% foreach my $item (@{$product->related}) { %>
    You might also enjoy <% $item->name %>.
<% } %>
```

And now Template Toolkit:

```
[% USE product(sku=bar1234) %]

[% IF product.isbn %]
    It's a book!
[% ELSE %]
    It's NOT a book!
[% END %]

[% FOREACH item = product.related %]
    You might also enjoy [% item.name %].
[% END %]
```

There is a third approach, based on parsing an HTML document into a DOM tree and then manipulating the contents of the nodes. The only module using this approach is `HTML_Tree`. The idea is similar to using a mini-language, but it doesn't require any non-standard HTML tags and it doesn't embed any logic about loops or conditionals in the template itself. This is nice because it means your templates are valid

HTML documents that can be viewed in a browser and worked with in most standard HTML tools. It also means people working with the templates can put placeholder data in them for testing and it will simply be replaced when the template is used. This preview ability only breaks down when you need an if/else type construct in the template. In that situation, both the "if" and "else" chunks of HTML would show up when previewing.

2.4.3 *Parsers and Caching*

The parsers for these templating systems are implemented in one of three ways: they parse the template every time ("repeated parse"), they parse it and cache the resulting parse tree ("cached parse tree"), or they parse it, convert it to Perl code, and compile it ("compiled").

Systems that compile templates to Perl take advantage of Perl's powerful runtime code evaluation capabilities. They examine the template, generate a chunk of Perl code from it, and `eval` the generated code. After that, subsequent requests for the template can be handled by running the compiled bytecode in memory. The complexity of the parsing and code generation steps varies based on the number of bells and whistles the system provides beyond straight in-line Perl statements.

Compiling to Perl and then to Perl bytecode is slow on the first hit but provides excellent performance once the template has been compiled, since the template becomes a Perl subroutine call. This is the same approach used by systems like JSP (Java ServerPages). It is most effective in environments with a long-running Perl interpreter, like `mod_perl`.

HTML::Template, HTML_Tree, and the 2.0 beta release of Embperl all use a cached parse tree approach. They parse templates into their respective internal data structures and then keep the parsed structure for each processed template in memory. This is similar to the compiled Perl approach in terms of performance and memory requirements, but does not actually involve Perl code generation and thus doesn't require an `eval` step. Which way is faster, caching the parse tree or compiling? It's hard to objectively measure, but anecdotal evidence seems to support compilation. Template Toolkit used a cached parse tree approach for version 1, but switched to a compilation approach for version 2 after tests showed it to offer a significant speed increase. However, as will be discussed later, either approach is more than fast enough.

In contrast to this, a repeated parse approach may sound very slow. However, it can be pretty fast if the tokens being parsed for are simple enough. Systems using this approach generally use very simple tokens, which allows them to use fast and simple parsers.

Why would you ever use a system with this approach if compilation has better performance? Well, in an environment without a persistent Perl interpreter like vanilla CGI this can actually be faster than a compiled approach since the startup cost is lower. The caching of Perl bytecode done by compilation systems is useless when the Perl interpreter doesn't stick around for more than one request.

There are other reasons too. Compiled Perl code takes up a lot of memory. If you have many unique templates, they can add up fast. Imagine how much RAM it would take up if every page that used server-side includes (SSI) had to stay in memory after it had been accessed. (Don't worry, the `Apache::SSI` module doesn't use compilation so it doesn't have this problem.)

2.4.4 Application Frameworks vs. Just Templates

Some of the templating tools try to offer a comprehensive solution to the problems of web development. Others offer just a templating solution and assume you will fit this together with other modules to build a complete system.

Some common features offered in the frameworks include:

2.4.4.1 URL Mapping

All of the frameworks offer a way to map a URL to a template file. In addition to simple mappings similar to the handling of static documents, some offer ways to intercept all requests within a certain directory for pre-processing, or create an object inheritance scheme out of the directory structure of a site.

2.4.4.2 Session Tracking

Most interactive sites need to use some kind of session tracking to associate application state data with a user. Some tools make this very easy by handling all the cookies or URL-munging for you and letting you simply read and write from an object or hash that contains the current user's session data. A common approach is to use the `Apache::Session` module for storage.

2.4.4.3 Output Caching

Caching is the key to good performance in many web systems, and some of these tools provide user-controlled caching of output. This is one of the major features of both `Mason` and `AxKit`. `AxKit` can cache at the page level, while `Mason` also offers fine-grained caching of components within the page.

2.4.4.4 Form Handling

How will you live without `CGI.pm` to parse incoming form data? Many of these tools will do it for you, making it available in a convenient data structure. Some also validate form input, and even provide "sticky" form widgets that keep their selected values when re-displayed or set up default values based on data you provide.

2.4.4.5 Debugging

Everyone knows how painful it can be to debug a CGI script. Templating systems can make it worse, by screwing up Perl's line numbers with generated code. To help fix the problem they've created, some offer built-in debugging support, including extra logging, or integration with the Perl debugger.

If you want to use a system that just does templates but you need some of these other features and don't feel like implementing them yourself, there are some tools on CPAN which provide a framework you can build on. The `libservlet` distribution, which provides an interface similar to the Java servlet API, is independent of any particular templating system. `Apache::PageKit` and `CGI::Application` are other options in this vein, but both of these are currently tied to `HTML::Template`. `OpenInteract` is another framework, this time tied to `Template Toolkit`. All of these could be adapted for the "just templates" module of your choice with fairly minimal effort.

2.5 The Contenders

Okay, now that you know something about what separates these tools from each other, let's take a look at the top choices for Perl templating systems. This is not an exhaustive list: I've only included systems that are currently maintained, well-documented, and have managed to build up a significant user community. In short, I've left out a dozen or so less popular systems. At the end of this section, I'll mention a few systems that aren't as commonly used but may be worth a look.

2.5.1 SSI

SSI is the granddaddy of templating systems, and the first one that many people used since it comes as a standard part of most web servers. With `mod_perl` installed, `mod_include` gains some additional power. Specifically, it is able to take a new `#perl` directive (though only if `mod_perl` is statically built) which allows for in-line subroutine calls. It can also efficiently include the output of `Apache::Registry` scripts by using the `Apache::Include` module.

The `Apache::SSI` module implements the functionality of `mod_include` entirely in Perl, including the additional `#perl` directive. The main reasons to use it are to post-process the output of another handler (with `Apache::Filter`) or to add your own directives. Adding directives is easy through subclassing. You might be tempted to implement a complete template processor in this way, by adding loops and other constructs, but it's probably not worth the trouble with so many other tools out there.

SSI follows the callback model and is mostly a mini-language, although you can sneak in bits of Perl code as anonymous subs in `#perl` directives. Because SSI uses a repeated parse implementation, it is safe to use it on large numbers of files without worrying about memory bloat.

SSI is a great choice for sites with fairly simple templating needs, especially ones that just want to share some standard headers and footers between pages. However, you should consider whether or not your site will eventually need to grow into something with more flexibility and power before settling on this simple approach.

2.5.2 *HTML::Mason*

Mason has been around for a few years now, and has built up a loyal following. It was originally created as a Perl clone of some of the most interesting features from Vignette StoryServer, but has since become it's own unique animal. It comes from a publishing background, and includes features oriented towards splitting up pages into re-useable chunks, or "components."

Mason uses in-line Perl with a compilation approach, but has a feature to help keep the perl code out of the HTML coder's way. Components (templates) can include a section of Perl at the end of the file which is wrapped inside a special tag indicating that it should be run first, before the rest of the template. This allows programmers to put all the logic for a component down at the bottom away from the HTML, and then use short in-line Perl snippets in the HTML to insert values, loop through lists, etc.

Mason is a site development framework, not just a templating tool. It includes a very handy caching feature that can be used for capturing the output of components or simply storing data that is expensive to compute. It is currently the only tool that offers this sort of caching as a built-in. It also implements an argument parsing scheme which allows a component to specify the names, types, and default values that it expects to be passed, either from another component or from the values passed in the URI query string.

While the documentation mostly demonstrates a callback execution model, it is possible to use Mason in a pipeline style. This can be accomplished in various ways, including designating components as "autohandlers" which run before anything else for requests within a certain directory structure. An autohandler could do some processing and set up data for a display template which only includes minimal in-line Perl. There is also support for an object-oriented site approach, applying concepts like inheritance to the site directory structure. For example, the component at `/store/book/` might inherit a standard layout from the component at `/store/`, but override the background color and navigation bar. Then `/store/music/` can do the same, with a different color. This can be a very powerful paradigm for developing large sites.

Mason's approach to debugging is to create "debug files" which run Mason outside of a web server environment, providing a fake web request and activating the debugger. This can be helpful if you're having trouble getting `Apache::DB` to behave under `mod_perl`, or using an execution environment that doesn't provide built-in debugger support.

Another unique feature is the ability to leave the static text parts of a large template on disk, and pull them in with a file seek when needed rather than keeping them in RAM. This exchanges some speed for a significant savings in memory when dealing with templates that are mostly static text.

There are many other features in this package, including filtering of HTML output and a page previewing utility. Session support is not built-in, but a simple example showing how to integrate with `Apache::Session` is included. Mason's feature set can be a bit overwhelming for newbies, but the high-quality documentation and helpful user community go a long way.

2.5.3 *HTML::Embperl*

Embperl makes its language choice known up front: embedded perl. It is one of the most popular in-line Perl templating tools and has been around longer than most of the others. It has a solid reputation for speed and ease of use.

It is commonly used in a callback style, with Embperl intercepting URIs and processing the requested file. However, it can optionally be invoked through a subroutine call from another program, allowing it to be used in a pipeline style. Templates are compiled to Perl bytecode and cached.

Embperl has been around long enough to build up an impressive list of features. It has the ability to run code inside a Safe compartment, support for automatically cleaning up globals to make `mod_perl` coding easier, and extensive debugging tools including the ability to e-mail errors to an administrator.

The main thing that sets Embperl apart from other in-line Perl systems is its tight HTML integration. It can recognize `TABLE` tags and automatically iterate over them for the length of an array. It automatically provides sticky form widgets. An array or hash reference placed at the end of a query string in an `HREF` or `SRC` attribute will be automatically expanded into query string `"name=value"` format. `META`

HTTP-EQUIV tags are turned into true HTTP headers.

Another reason people like Embperl is that it makes some of the common tasks of web application coding so simple. For example, all form data is always available just by reading the magic variable %fdat. Sessions are supported just as easily, by reading and writing to the magic %udat hash. There is also a hash for storing persistent application state. HTML-escaping is automatic (though it can be toggled on and off).

Embperl includes something called EmbperlObject, which allows you to apply OO concepts to your site hierarchy in a similar way to the inheritance features mentioned for Mason, above. This is a very convenient way to code sites with styles that vary by area, and is worth checking out.

One drawback of older versions of Embperl was the necessity to use built-in replacements for most of Perl's control structures like "if" and "foreach" when they are being wrapped around non-Perl sections. For example:

```
[$ if ($foo) $]
    Looks like a foo!
[$ else $]
    Nope, it's a bar.
[$ endif $]
```

These may seem out of place in a system based around in-line Perl. As of version 1.2b2, it is possible to use Perl's standard syntax instead:

```
[$ if ($foo) { $]
    Looks like a foo!
[$ } else { $]
    Nope, it's a bar.
[$ } $]
```

At the time of this writing, a new 2.x branch of Embperl is in beta testing. This includes some interesting features like a more flexible parsing scheme which can be modified to users' tastes. It also supports direct use of the Perl debugger on Embperl templates, and provides performance improvements.

2.5.4 *Apache::AxKit*

AxKit is the first mod_perl page generation system to be built from the ground up around XML. Technically, AxKit itself is not a templating tool but rather a framework for stringing together different modules that generate and transform XML data. In fact, it can optionally use Template Toolkit as an XML transformation language. However, it deserves coverage here since it is also the home of some templating tools that are not represented elsewhere.

In its simplest form, AxKit maps XML files to XSL stylesheets which it can process using commonly available XSLT modules like XML::XSLT or XML::Sabltron. The rules for mapping a stylesheet to a request are very flexible, and they can incorporate query strings, cookies, and other attributes of the request. The idea is that you can use this feature to handle a wide variety of clients with differing display capabilities by choosing the right stylesheet.

Recognizing that not everyone is a fan of XSL's somewhat obtuse syntax, Matt Sergeant has provided an alternate stylesheet language called XPathScript. XPathScript allows you to write a stylesheet using text with embedded Perl code. This is similar to the other embedded Perl templating tools, but the focus is on using the built in XPath functions for querying an XML document and manipulating the retrieved data. XPathScript can also be used in a declarative fashion, specifying the formatting of particular elements in the XML input. For example this snippet will change all <foo> tags in an XML document to BAR in the output::

```
<%
  $t->{'foo'}{pre}    = 'BAR';
  $t->{'foo'}{post}    = '';
  $t->{'foo'}{showtag} = 0;
%>
<%= apply_templates() %>
```

By using XPathScript's include function (which looks just like SSI), you can build up libraries of useful transformations that use this technique.

This is all well and good if you have a bunch of XML files sitting on a disk somewhere, but what about dynamic content? AxKit handles this by allowing you to substitute a different data source for the default file-based one. This can include running some dynamic code on each request to generate the XML data that will be transformed. The distribution includes a module for doing this called XSP. XSP is a language for building an XML DOM using in-line Perl and tag libraries. The tag libraries are specified as stylesheets which can turn XML tags into Perl code. This is demonstrated through the included SQL tag library, which allows you to write an XSP page using XML tags which will connect to a database, execute queries, and generate an XML document with the results.

AxKit has some nice performance boosts built into it. It can cache the full output of a page and serve it as a static file on future requests. It can also compress output to speed up downloads for browsers that understand gzip encoding. These can be done with other systems, but they require you to setup additional software. With AxKit, you just enable them in the configuration file.

If all of these languages, tag libraries, and stylesheets sound intimidating to you, AxKit may be overkill for your project. However, AxKit has the advantage of being built on approved W3C standards, and many of the skills used in developing for it carry over to other languages and tools.

2.5.5 Apache::ASP

Apache::ASP started out as a port of Microsoft's Active Server Pages technology, and its basic design still follows that model. It uses in-line Perl with a compilation approach, and provides a set of simple objects for accessing the request information and formulating a response. Scripts written for Microsoft's ASP using Perl (via ActiveState's PerlScript) can usually be run on this system without changes. (Pages written in VBScript are not supported.)

Like the original ASP, it has hooks for calling specified code when certain events are triggered, such as the start of a new user session. It also provides the same easy-to-use state and session management. Storing and retrieving state data for a whole application or a specific user is as simple as a single method call. It can even support user sessions without cookies by munging URLs -- a unique feature among these

systems.

A significant addition that did not come from Microsoft ASP is the XML and XSLT support. There are two options provided: XMLSubs and XSLT transforms. XMLSubs is a way of adding custom tags to your pages. It maps XML tags to your subroutines, so that you can add something like `<site:header page="Page Title" />` to your pages and have it translate into a subroutine call like `&site::header({title => "Page Title"})`. It can handle processing XML tags with body text as well.

The XSLT support allows the output of ASP scripts to be filtered through XSLT for presentation. This allows your ASP scripts to generate XML data and then format that data with a separate XSL stylesheet. This support is provided through integration with the XML::XSLT module.

Apache::ASP provides sticky widgets for forms through the use of the HTML::FillInForm module. It also has built-in support for removing extra whitespace from generated output, gzip compressing output (for browsers that support it), tracking performance using Time::HiRes, automatically mailing error messages to an administrator, and many other conveniences and tuning options. This is a mature package which has evolved to handle real-world problems.

One thing to note about the session and state management in this system is that it currently only supports clusters through the use of network filesystems like NFS or SMB. (Joshua Chamas, the module's author, has reported much better results from Samba file-sharing than from NFS.) This may be an issue for large-scale server clusters, which usually rely on a relational database for network storage of sessions. Support database storage of sessions is planned for a future release.

2.5.6 *Text::Template*

This module has become the de facto standard general purpose templating module on CPAN. It has an easy interface and thorough documentation. The examples in the docs show a pipeline execution style, but it's easy to write a `mod_perl` handler that directly invokes templates, allowing a callback style. The module uses in-line Perl. It has the ability to run the in-line code in a Safe compartment, in case you are concerned about mistakes in the code crashing your server.

The module relies on creative uses of in-line code to provide things that people usually expect from templating tools, like includes. This can be good or bad. For example, to include a file you could just call `Text::Template::fill_in_file(filename)`. However, you'll have to specify the complete file path and nothing will stop you from using `/etc/passwd` as the file to be included. Most of the fancier templating tools have concepts like include paths, which allow you to specify a list of directories to search for included files. You could write a subroutine that works this way, and make it available in your template's namespace, but it's not built in.

Each template is loaded as a separate object. Templates are compiled to Perl and only parsed the first time they are used. However, to take full advantage of this caching in a persistent environment like `mod_perl`, your program will have to keep track of which templates have been used, since `Text::Template` does not have a way of globally tracking this and returning cached templates when possible.

Text::Template is not tied to HTML, and is just a templating module, not a web application framework. It is perfectly at home generating e-mails, PDFs, etc.

2.5.7 *Template Toolkit*

One of the more recent additions to the templating scene, Template Toolkit is a very flexible mini-language system. It has a complete set of directives for working with data, including loops and conditionals, and it can be extended in a number of ways. In-line Perl code can be enabled with a configuration option, but is generally discouraged. It uses compilation, caching the compiled bytecode in memory and optionally caching the generated Perl code for templates on disk. Although it is commonly used in a pipeline style, the included Apache::Template module allows templates to be invoked directly from URLs.

Template Toolkit has a large feature set, so we'll only be able cover some of the highlights here. The TT distribution sets a gold standard for documentation thoroughness and quality, so it's easy to learn more if you choose to.

One major difference between TT and other systems is that it provides simple access to complex data structures through the concept of a dot operator. This allows people who don't know Perl to access nested lists and hashes or call object methods. For example, we could pass in this Perl data structure:

```
$vars = {
    customer => {
        name    => 'Bubbles',
        address => {
            city => 'Townsville',
        }
    }
};
```

Then we can refer to the nested data in the template:

```
Hi there, [% customer.name %]!
How are things in [% customer.address.city %]?
```

This is simpler and more uniform than the equivalent syntax in Perl. If we pass in an object as part of the data structure, we can use the same notation to call methods within that object. If you've modeled your system's data as a set of objects, this can be very convenient.

Templates can define macros and include other templates, and parameters can be passed to either. Included templates can optionally localize their variables so that changes made while the included template is executing do not affect the values of variables in the larger scope.

There is a filter directive, which can be used for post-processing output. Uses for this range from simple HTML entity conversion to automatic truncation (useful for pulldown menus when you want to limit the size of entries) and printing to STDERR.

TT supports a plugin API, which can be used to add extra capabilities to your templates. The provided plugins can be broadly organized into data access and formatting. Standard data access plugins include modules for accessing XML data or a DBI data source and using that data within your template. There's a plugin for access to CGI.pm as well.

Formatting plugins allow you to display things like dates and prices in a localized style. There's also a table plugin for use in displaying lists in a multi-column format. These formatting plugins do a good job of covering the final 5% of data display problems that often cause people who are using an in-house system to embed a little bit of HTML in their Perl modules.

In a similar vein, TT includes some nice convenience features for template writers like eliminating white space around tags and the ability to change the tag delimiters -- things that may sound a little esoteric, but can sometimes make templates significantly easier to work with.

The TT distribution also includes a script called ttree which allows for processing an entire directory tree of templates. This is useful for sites that pre-publish their templated pages and serve them statically. The script checks modification times and only updates pages that require it, providing a make-like functionality. The distribution also includes a sample set of template-driven HTML widgets which can be used to give a consistent look and feel to a collection of documents.

2.5.8 *HTML::Template*

HTML::Template is a popular module among those looking to use a mini-language rather than in-line Perl. It uses a simple set of tags which allow looping (even on nested data structures) and conditionals in addition to basic value insertion. The tags are intentionally styled to look like HTML tags, which may be useful for some situations.

As the documentation says, it "does just one thing and it does quickly and carefully" -- there is no attempt to add application features like form-handling or session tracking. The module follows a pipeline execution style. Parsed templates are stored in a Perl data structure which can be cached in any combination of memory, shared memory (using IPC::SharedCache), and disk. The documentation is complete and well-written, with plenty of examples.

You may be wondering how this module is different from Template Toolkit, the other popular mini-language system. Beyond the obvious differences in syntax, HTML::Template is faster and simpler, while Template Toolkit has more advanced features, like plugins and dot notation. Here's a simple example comparing the syntax:

HTML::Template:

```
<TMPL_LOOP list>
  <a href=<TMPL_VAR url>><b><TMPL_VAR name></b></a>
</TMPL_LOOP>
```

Template Toolkit:

```
[% FOREACH list %]
  <a href="[% url %]"><b>[% name %]</a></a>
[% END %]
```

And now, a few honorable mentions:

2.5.9 *HTML_Tree*

As mentioned earlier, HTML Tree uses a fairly unique method of templating: it loads in an HTML page, parses it to a DOM, and then programmatically modifies the contents of nodes. This allows it to use genuine valid HTML documents as templates, something which none of these other modules can do. The learning curve is a little steeper than average, but this may be just the thing if you are concerned about keeping things simple for your HTML coders. Note that the name is "HTML_Tree", not "HTML::Tree".

2.5.10 *Apache::XPP*

XPP is an in-line Perl system that compiles to bytecode. Although it is a perfectly good implementation, it has little to differentiate it except for an easy mechanism to define new HTML-like tags which can be used to replace in-line code in templates.

2.5.11 *ePerl*

Possibly the first module to embed Perl code in a text or HTML file, ePerl is still a viable option in the form of Apache::ePerl. It caches compiled bytecode in memory to achieve solid performance, and some people find it refreshingly simple to use.

2.5.12 *CGI::FastTemplate*

This module takes a minimalistic approach to templating, which makes it unusually well suited to use in CGI programs. It parses templates with a single regular expression and does not support anything in templates beyond simple variable interpolation. Loops are handled by including the output of other templates. Unfortunately, this leads to a Perl coding style that is more confusing than most, and a proliferation of template files. However, some people swear by this dirt-simple approach.

2.6 Performance

People always seem to worry about the performance of templating systems. If you've ever built a large-scale application, you should have enough perspective on the relative costs of different actions to know that your templating system is not the first place to look for performance gains. All of the systems mentioned here have excellent performance characteristics in persistent execution environments like mod_perl. Compared to such glacially slow operations as fetching data from a database or file, the time added by the templating system is almost negligible.

If you think your templating system is slowing you down, get the facts: pull out Devel::DProf and see. If one of the tools mentioned here is at the top of the list for wall clock time used, you should pat yourself on the back -- you've done a great job tuning your system and removing bottlenecks! Personally, I have only seen this happen when I had managed to successfully cache nearly every part of the work to handle a request except running a template.

However, if you really are in a situation where you need to squeeze a few extra microseconds out of your page generation time, there are performance differences between systems. They're pretty much what you would expect: systems that do the least run the fastest. Using in-line `print()` statements is faster than using templates. Using simple substitution is faster than using in-line Perl code. Using in-line Perl code is faster than using a mini-language.

The only templating benchmark available at this time is one developed by Joshua Chamas, author of `Apache::ASP`. It includes a "hello world" test, which simply checks how fast each system can spit back those famous words, and a "hello 2000" test, which exercises the basic functions used in most dynamic pages. It is available from the following URL:

<http://www.chamas.com/bench/hello.tar.gz>

Results from this benchmark currently show SSI, `Apache::ASP`, and `HTML::Embperl` having the best performance of the lot. Not all of the systems mentioned here are currently included in the test. If your favorite was missed, you might want to download the benchmark code and add it. As you can well imagine, benchmarking people's pet projects is largely a thankless task and Joshua deserves some recognition and support for this contribution to the community.

2.6.1 CGI Performance Concerns

If you're running under CGI, you have bigger fish to fry than worrying about the performance of your templating system. Nevertheless, some people are stuck with CGI but still want to use a templating system with reasonable performance. CGI is a tricky situation, since you have to worry about how much time it will take for Perl to compile the code for a large templating system on each request. CGI also breaks the in-memory caching of templates used by most of these systems, although the slower disk-based caching provided by `Mason`, `HTML::Template`, and `Template Toolkit` will still work. (`HTML::Template` does provide a shared memory cache for templates, which may improve performance, although shared memory on my Linux system is usually slower than using the filesystem. Benchmarks and additional information are welcome.)

Your best performance bet with CGI is to use one of the simpler tools, like `CGI::FastTemplate` or `Text::Template`. They are small and compile quickly, and `CGI::FastTemplate` gets an extra boost since it relies on simple regex parsing and doesn't need to eval any in-line Perl code. Almost everything else mentioned here will add tenths of seconds to each page in compilation time alone.

2.7 Matrix

To help you choose a system, I'll summarize the basic characteristics of the major systems along the decision points I've explained in the beginning of the article. Keep in mind that in many cases a system can be used in more than one way, and I've simply shown the dominant method as seen in the documentation and real world use. You should not eliminate options based on this chart without reading the more detailed explanations above.

	Application Framework	Pipeline or Callback	Parsing Method	Language
HTML::Mason	Framework	Callback	Compiled	Perl
Template Toolkit	Just Templates	Pipeline	Compiled	Mini-Language
Apache::ASP	Framework	Callback	Compiled	Perl and XSL
HTML::Embperl	Framework	Callback	Compiled	Perl
SSI	Just Templates	Callback	Repeated Parse	Mini-Language
AxKit	Framework	Pipeline	Compiled or Cached Parse Tree	Perl and XSL and Mini-Language(s)
HTML::Template	Just Templates	Pipeline	Cached Parse Tree	Mini-Language
Text::Template	Just Templates	Pipeline	Compiled	Perl

2.8 Updates

These modules are moving targets, and a document like this is bound to contain some mistakes. Send your corrections to perrin@elem.com. Future versions of this document will be announced on the `mod_perl` mailing list, and possibly other popular Perl locations as well.

by Perrin Harkins

2.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Perrin Harkins <[perrin \(at\) elem.com](mailto:perrin@elem.com)>.

2.10 Authors

- Perrin Harkins <[perrin \(at\) elem.com](mailto:perrin@elem.com)>.

Only the major authors are listed above. For contributors see the Changes file.

3 Cute Tricks With Perl and Apache

3.1 Description

Perl and Apache play very well together, both for administration and coding. However, adding `mod_perl` to the mix creates a heaven for an administrator/programmer wanting to do cool things in no time!

This tutorial begins a collection of CGI scripts that illustrate the three basic types of CGI scripting: dynamic documents, document filtering, and URL redirection. It also shows a few tricks that you might not have run into -- or even thought were possible with CGI.

Then, we move to look at different uses of Perl to handle typical administrative tasks. Finally, we continue with the next step beyond CGI scripting: the creation of high performance Apache modules with the `mod_perl` API.

3.2 Part I: Tricks with CGI.pm

`CGI.pm` is the long-favoured module for CGI scripting, and, as `mod_perl` can run CGI scripts (mostly) unaltered, also provides significant advantages for `mod_perl` programmers. Let's look at some of the more interesting uses of this module in web programming.

3.2.1 *Dynamic Documents*

The most familiar use of CGI is to create documents on the fly. They can be simple documents, or get incredibly baroque. We won't venture much past the early baroque.

3.2.1.1 Making HTML look beautiful

`<I>` `<hate>` `<HTML>` `<because>` `<it's>` `<ugly>` `<and>` `<has>` `<too>` `<many>` `<$#@*>` `<angle>` `<bracket>`. With `CGI.pm` it's almost good to look at. Script I.1.1 shows what a nested list looks like with `CGI.pm`.

```
Script I.1.1: vegetables1.pl
-----
#!/usr/bin/perl
# Script: vegetables1.pl
use CGI::Pretty ':standard';
print header,
    start_html('Vegetables'),
    h1('Eat Your Vegetables'),
    ol(
        li('peas'),
        li('broccoli'),
        li('cabbage'),
        li('peppers',
            ul(
                li('red'),
                li('yellow'),
                li('green')
            )
        ),
        li('kolrabi'),
```

```

        li('radishes')
    ),
    hr,
    end_html;

```

3.2.1.2 Making HTML concise

But we can do even better than that because CGI.pm lets you collapse repeating tags by passing array references to its functions. Script 1.2 saves some typing, and in so doing, puts off the onset of RSI by months or years!

```

Script I.1.2: vegetables2.pl
-----
#!/usr/bin/perl
# Script: vegetables2.pl
use CGI ':standard';
print header,
    start_html('Vegetables'),
    h1('Eat Your Vegetables'),
    ol(
        li([
            'peas',
            'broccoli',
            'cabbage',
            'peppers' .
            ul(['red', 'yellow', 'green']),
            'kolrabi',
            'radishes'
        ]),
        hr,
        end_html;

```

Or how about this one?

```

Script I.1.3: vegetables3.pl
-----
#!/usr/bin/perl

# Script: vegetables3.pl
use CGI::Pretty qw/:standard :html3/;

print header,
    start_html('Vegetables'),
    h1('Vegetables are for the Strong'),
    table({-border=>undef},
        caption(strong('When Should You Eat Your Vegetables?')),
        Tr({-align=>CENTER, -valign=>TOP},
            [
                th(['', 'Breakfast', 'Lunch', 'Dinner']),
                th('Tomatoes').td(['no', 'yes', 'yes']),
                th('Broccoli').td(['no', 'no', 'yes']),
                th('Onions').td(['yes', 'yes', 'yes'])
            ]
        )
    ),
    end_html;

```


3.2.1.3 Making Interactive Forms

Of course you mostly want to use CGI to create interactive forms. No problem! CGI.pm has a full set of functions for both generating the form and reading its contents once submitted. Script I.1.4 creates a row of radio buttons labeled with various colors. When the user selects a button and submits the form, the page redraws itself with the selected background color. Psychedelic!

```
Script I.1.4: customizable.pl
-----
#!/usr/bin/perl
# script: customizable.pl

use CGI::Pretty qw/:standard/;

my $color = param('color') || 'white';

print header,
    start_html({-bgcolor=>$color}, 'Customizable Page'),
    h1('Customizable Page'),
    "Set this page's background color to:", br,
    start_form,
    radio_group(-name=>'color',
                -value=>['white', 'red', 'green', 'black',
                        'blue', 'silver', 'cyan'],
                -cols=>2),
    submit(-name=>'Set Background'),
    end_form,
    p,
    hr,
    end_html;
```

3.2.2 Making Stateful Forms

Many real Web applications are more than a single page. Some may span multiple pages and fill-out forms. When the user goes from one page to the next, you've got to save the state of the previous page somewhere. A convenient and cheap place to put state information is in hidden fields in the form itself. Script I.2.1 is an example of a loan application with a total of five separate pages. Forward and back buttons allows the user to navigate between pages. The script remembers all the pages and summarizes them up at the end.

```
Script I.2.1: loan.pl
-----
#!/usr/local/bin/perl

# script: loan.pl
use CGI qw/:standard :html3/;

# this defines the contents of the fill out forms
# on each page.
my @PAGES = ('Personal Information', 'References', 'Assets', 'Review', 'Confirmation');
my %FIELDS = ('Personal Information' => ['Name', 'Address', 'Telephone', 'Fax'],
              'References'           => ['Personal Reference 1', 'Personal Reference 2'],
              'Assets'               => ['Savings Account', 'Home', 'Car']
              );
```

3.2.2 Making Stateful Forms

```
my %ALL_FIELDS = ();
# accumulate the field names into %ALL_FIELDS;
foreach (values %FIELDS) {
    grep(%ALL_FIELDS{$_}++, @$_);
}

# figure out what page we're on and where we're heading.
my $current_page = calculate_page(param('page'),param('go'));
my $page_name = $PAGES[$current_page];

print_header($page_name);
print_form($current_page)      if $FIELDS{$page_name};
print_review($current_page)    if $page_name eq 'Review';
print_confirmation($current_page) if $page_name eq 'Confirmation';
print end_html;

# CALCULATE THE CURRENT PAGE
sub calculate_page {
    my ($prev, $dir) = @_;
    return 0 if $prev eq '';          # start with first page
    return $prev + 1 if $dir eq 'Submit Application';
    return $prev + 1 if $dir eq 'Next Page';
    return $prev - 1 if $dir eq 'Previous Page';
}

# PRINT HTTP AND HTML HEADERS
sub print_header {
    my $page_name = shift;
    print header,
        start_html("Your Friendly Family Loan Center"),
        h1("Your Friendly Family Loan Center"),
        h2($page_name);
}

# PRINT ONE OF THE QUESTIONNAIRE PAGES
sub print_form {
    my $current_page = shift;
    print "Please fill out the form completely and accurately.",
        start_form,
        hr;
    draw_form(@{$FIELDS{$page_name}});
    print hr;
    print submit(-name=>'go',-value=>'Previous Page')
        if $current_page > 0;
    print submit(-name=>'go',-value=>'Next Page'),
        hidden(-name=>'page',-value=>$current_page,-override=>1),
        end_form;
}

# PRINT THE REVIEW PAGE
sub print_review {
    my $current_page = shift;
    print "Please review this information carefully before submitting it. ",
        start_form;
    my (@rows);
    foreach $page ('Personal Information','References','Assets') {
        push(@rows,th({-align=>LEFT},em($page)));
    }
}
```

```

        foreach $field (@{$FIELDS{$page}}) {
            push(@rows,
                TR(th({-align=>LEFT},$field),
                    td(param($field)))
                );
            print hidden(-name=>$field);
        }

print table({-border=>1},caption($page),@rows),
    hidden(-name=>'page',-value=>$current_page,-override=>1),
    submit(-name=>'go',-value=>'Previous Page'),
    submit(-name=>'go',-value=>'Submit Application'),
end_form;
}

# PRINT THE CONFIRMATION PAGE
sub print_confirmation {
    print "Thank you. A loan officer will be contacting you shortly.",
        p,
        a({-href=>'../source.html'},'Code examples');
}

# CREATE A GENERIC QUESTIONNAIRE
sub draw_form {
    my (@fields) = @_;
    my (%fields);
    grep ($fields{$_}++, @fields);
    my (@hidden_fields) = grep(!$fields{$_}, keys %ALL_FIELDS);
    my (@rows);
    foreach (@fields) {
        push(@rows,
            TR(th({-align=>LEFT},$_),
                td(textfield(-name=>$_,-size=>50))
                )
            );
    }
    print table(@rows);

    foreach (@hidden_fields) {
        print hidden(-name=>$_);
    }
}

```

3.2.2.1 Keeping State with Cookies

If you want to maintain state even if the user quits the browser and comes back again, you can use cookies. Script I.2.2 records the user's name and color scheme preferences and recreates the page the way the user likes up to 30 days from the time the user last used the script.

```

Script I.2.2: preferences.pl
-----
#!/usr/local/bin/perl

# file: preferences.pl

```

3.2.2 Making Stateful Forms

```
use CGI qw(:standard :html3);

# Some constants to use in our form.
my @colors = qw/aqua black blue fuchsia gray green lime maroon navy olive
  purple red silver teal white yellow/;
my @sizes=("<default>",1..7);

# recover the "preferences" cookie.
my %preferences = cookie('preferences');

# If the user wants to change the background color or her
# name, they will appear among our CGI parameters.
foreach ('text','background','name','size') {
    $preferences{$_} = param($_) || $preferences{$_};
}

# Set some defaults
$preferences{'background'} ||= 'silver';
$preferences{'text'} ||= 'black';

# Refresh the cookie so that it doesn't expire.
my $the_cookie = cookie(-name=>'preferences',
                        -value=>\%preferences,
                        -path=>'/',
                        -expires=>'+30d');
print header(-cookie=>$the_cookie);

# Adjust the title to incorporate the user's name, if provided.
$title = $preferences{'name'} ?
    "Welcome back, $preferences{name}!" : "Customizable Page";

# Create the HTML page. We use several of the HTML 3.2
# extended tags to control the background color and the
# font size. It's safe to use these features because

# cookies don't work anywhere else anyway.
print start_html(-title=>$title,
                -bgcolor=>$preferences{'background'},
                -text=>$preferences{'text'}
                );

print basefont({-size=>$preferences{size}}) if $preferences{'size'} > 0;

print h1($title);

# Create the form
print hr,
    start_form,

    "Your first name: ",
    textfield(-name=>'name',
              -default=>$preferences{'name'},
              -size=>30),br,

    table(
        TR(
```

```

        td("Preferred"),
        td("Page color:"),
        td(popup_menu(-name=>'background',
                      -values=>\@colors,
                      -default=>$preferences{'background'}))
    ),
),
TR(
    td(''),
    td("Text color:"),
    td(popup_menu(-name=>'text',
                  -values=>\@colors,
                  -default=>$preferences{'text'}))
)
),
TR(
    td(''),
    td("Font size:"),
    td(popup_menu(-name=>'size',
                  -values=>\@sizes,
                  -default=>$preferences{'size'}))
)
),
),
submit(-label=>'Set preferences'),
end_form,
hr,
end_html;

```

3.2.3 *Creating Non-HTML Types*

CGI can do more than just produce HTML documents. It can produce any type of document that you can output with Perl. This includes GIFs, Postscript files, sounds or whatever.

Script I.3.1 creates a clickable image map of a colored circle inside a square. The script is responsible both for generating the map and making the image (using the GD.pm library). It also creates a fill-out form that lets the user change the size and color of the image!

```

Script I.3.1: circle.pl
-----
#!/usr/local/bin/perl

# script: circle.pl
use GD;
use CGI qw/:standard :imagemap/;

use constant RECTSIZE    => 100;
use constant CIRCLE_RADIUS => 40;
my %COLORS = (
    'white' => [255,255,255],
    'red'   => [255,0,0],
    'green' => [0,255,0],
    'blue'  => [0,0,255],

```

3.2.3 Creating Non-HTML Types

```
'black' => [0,0,0],
'bisque'=> [255,228,196],
'papaya whip' => [255,239,213],
'sienna' => [160,82,45]
);

my $draw          = param('draw');
my $circle_color  = param('color') || 'bisque';
my $mag           = param('magnification') || 1;

if ($draw) {
    draw_image();
} else {
    make_page();
}

sub draw_image {
    # create a new image
    my $im = new GD::Image(RECTSIZE*$mag,RECTSIZE*$mag);

    # allocate some colors
    my $white = $im->colorAllocate(@{$COLORS{'white'}});
    my $black = $im->colorAllocate(@{$COLORS{'black'}});
    my $circlecolor = $im->colorAllocate(@{$COLORS{$circle_color}});

    # make the background transparent and interlaced
    $im->transparent($white);
    $im->interlaced('true');

    # Put a black frame around the picture
    $im->rectangle(0,0,RECTSIZE*$mag-1,RECTSIZE*$mag-1,$black);

    # Draw the circle
    $im->arc(RECTSIZE*$mag/2,RECTSIZE*$mag/2,
            CIRCLE_RADIUS*$mag*2,
            CIRCLE_RADIUS*$mag*2,
            0,360,$black);

    # And fill it with circlecolor
    $im->fill(RECTSIZE*$mag/2,RECTSIZE*$mag/2,$circlecolor);

    # Convert the image to GIF and print it
    print header('image/gif'),$im->gif;
}

sub make_page {
    print header(),
    start_html(-title=>'Feeling Circular',-bgcolor=>'white'),
    h1('A Circle is as a Circle Does'),
    start_form,
    "Magnification: ",radio_group(-name=>'magnification',-values=>[1..4]),br,
    "Color: ",popup_menu(-name=>'color',-values=>[sort keys %COLORS]),
    submit(-value=>'Change'),
    end_form;
    print em(param('message') || 'click in the drawing' );

    my $url = url(-relative=>1,-query_string=>1);
```

```

$url .= '?' unless param();
$url .= '&draw=1';

print p(
    img({-src=>$url,
        -align=>'LEFT',
        -usemap=>'#map',
        -border=>0}));

print Map({-name=>'map',
    Area({-shape=>'CIRCLE',
        -href=>param(-name=>'message',-value=>"You clicked in the circle")
        && url(-relative=>1,-query_string=>1),
        -coords=>join(',',RECTSIZE*$mag/2,RECTSIZE*$mag/2,CIRCLE_RADIUS*$mag),
        -alt=>'Circle'}),
    Area({-shape=>'RECT',
        -href=>param(-name=>'message',-value=>"You clicked in the square")
        && url(-relative=>1,-query_string=>1),
        -coords=>join(',',0,0,RECTSIZE*$mag,RECTSIZE*$mag),
        -alt=>'Square'}));
print end_html;
}

```

Script I.3.2 creates a GIF89a animation. First it creates a set of simple GIFs, then uses the *combine* program (part of the ImageMagick package) to combine them together into an animation.

I'm not a good animator, so I can't do anything fancy. But you can!

```

Script I.3.2: animate.pl
-----
#!/usr/local/bin/perl

# script: animated.pl
use GD;
use File::Path;

use constant START      => 80;
use constant END        => 200;
use constant STEP       => 10;
use constant COMBINE    => '/usr/local/bin/convert';
my @COMBINE_OPTIONS = (-delay => 5,
                      -loop  => 10000);

my @COLORS = ([240,240,240],
              [220,220,220],
              [200,200,200],
              [180,180,180],
              [160,160,160],
              [140,140,140],
              [150,120,120],
              [160,100,100],
              [170,80,80],
              [180,60,60],
              [190,40,40],
              [200,20,20],
              [210,0,0]);

```

3.2.4 Document Translation

```
@COLORS = (@COLORS,reverse(@COLORS));

my @FILES = ();
my $dir = create_temporary_directory();
my $index = 0;
for (my $r = START; $r <= END; $r+=STEP) {
    draw($r,$index,$dir);
    $index++;
}
for (my $r = END; $r > START; $r-=STEP) {
    draw($r,$index,$dir);
    $index++;
}

# emit the GIF89a
$| = 1;
print "Content-type: image/gif\n\n";
system COMBINE,@COMBINE_OPTIONS,@FILES,"gif:-";

rmtree([$dir],0,1);

sub draw {
    my ($r,$color_index,$dir) = @_;
    my $im = new GD::Image(END,END);
    my $white = $im->colorAllocate(255,255,255);
    my $black = $im->colorAllocate(0,0,0);
    my $color = $im->colorAllocate(@{$COLORS[$color_index % @COLORS]});
    $im->rectangle(0,0,END,END,$white);
    $im->arc(END/2,END/2,$r,$r,0,360,$black);
    $im->fill(END/2,END/2,$color);
    my $file = sprintf("%s/picture.%02d.gif",$dir,$color_index);
    open (OUT,">$file") || die "couldn't create $file: $!";
    print OUT $im->gif;
    close OUT;
    push(@FILES,$file);
}

sub create_temporary_directory {
    my $basename = "/usr/tmp/animate$$";
    my $counter=0;
    while ($counter < 100) {
        my $try = sprintf("$basename.%04d",$counter);
        next if -e $try;
        return $try if mkdir $try,0700;
    } continue { $counter++; }
    die "Couldn't make a temporary directory";
}
```

3.2.4 Document Translation

Did you know that you can use a CGI script to translate other documents on the fly? No s**t! Script I.4.1 is a script that intercepts all four-letter words in text documents and stars out the naughty bits. The document itself is specified using additional path information. We're a bit over-literal about what a four-letter word is, but what's the fun if you can't be extravagant?

Script I.4.1: naughty.pl

```
#!/usr/local/bin/perl
# Script: naughty.pl

use CGI ':standard';
my $file = path_translated() ||
    die "must be called with additional path info";
open (FILE,$file) || die "Can't open $file: $!\n";
print header('text/plain');
while (<FILE>) {
    s/\b(\w)\w{2}(\w)\b/$1**$2/g;
    print;
}
close FILE;
```

4.1 won't work on HTML files because the HTML tags will get starred out too. If you find it a little limiting to work only on plain-text files, script I.4.2 uses LWP's HTML parsing functions to modify just the text part of an HTML document without touching the tags. The script's a little awkward because we have to guess the type of file from the extension, and *redirect* when we're dealing with a non-HTML file. We can do better with *mod_perl*.

Script I.4.2: naughty2.pl

```
-----
#!/usr/local/bin/perl

# Script: naughty2.pl
package HTML::Parser::FixNaughty;

require HTML::Parser;
@HTML::Parser::FixNaughty::ISA = 'HTML::Parser';

sub start {
    my ($self,$tag,$attr,$attrseq,$origtext) = @_;
    print $origtext;
}
sub end {
    my ($self,$tag) = @_;
    print "</$tag>";
}
sub text {
    my ($self,$text) = @_;
    $text =~ s/\b(\w)\w{2}(\w)\b/$1**$2/g;
    print $text;
}

package main;
use CGI qw/header path_info redirect path_translated/;

my $file = path_translated() ||
    die "must be called with additional path info";
$file .= "index.html" if $file =~ m!/$!;

unless ($file =~ /\.html?$/) {
    print redirect(path_info());
}
```

```

        exit 0;
    }

    my $parser = new HTML::Parser::FixNaughty;
    print header();
    $parser->parse_file($file);

```

A cleaner way to do this is to make this into an Apache Handler running under mod_perl. Let's look at Apache::FixNaughty:

```

file:Apache/FixNaughty.pm
-----
# prefefine the HTML parser that we use afterwards
package HTML::Parser::FixNaughty;

require HTML::Parser;
@HTML::Parser::FixNaughty::ISA = 'HTML::Parser';

sub start {
    my ($self,$tag,$attr,$attrseq,$origtext) = @_;
    print $origtext;
}
sub end {
    my ($self,$tag) = @_;
    print "</$tag>";
}
sub text {
    my ($self,$text) = @_;
    $text =~ s/\b(\w)\w{2}(\w)\b/$1**$2/g;
    print $text;
}

# now for the mod_perl handler
package Apache::FixNaughty;

use Apache::Constants qw/:common/;
use strict;
use warnings;
use CGI qw/header path_info redirect path_translated/;

sub handler {
    my $r = shift;

    unless(-e $r->finfo) {
        $r->log_reason("Can't be found", $r->filename);
        return NOT_FOUND;
    }

    unless ($r->content_type eq 'text/html') {
        return DECLINED;
    }

    my $parser = new HTML::Parser::FixNaughty;

    $r->send_http_header('text/html');
    $parser->parse_file($file);

```

```

        return OK;
    }

    1;
__END__

```

You'll configure this like so:

```

Alias /naughty/ /path/to/doc/root/
<Location /naughty>
    SetHandler perl-script
    PerlHandler Apache::FixNaughty
</Location>

```

Now, all files being served below the */naughty* URL will be the same as those served from your document root, but will be processed and censored!

3.2.4.1 Smart Redirection

There's no need even to create a document with CGI. You can simply *redirect* to the URL you want. Script I.4.3 chooses a random picture from a directory somewhere and displays it. The directory to pick from is specified as additional path information, as in:

```

        /cgi-bin/random_pict/banners/egregious_advertising

Script I.4.3 random_pict.pl
-----
#!/usr/local/bin/perl
# script: random_pict.pl

use CGI qw(:standard);
my $PICTURE_PATH = path_translated();
my $PICTURE_URL = path_info();
chdir $PICTURE_PATH
    or die "Couldn't chdir to pictures directory: $!";
my @pictures = <*. {jpg,gif}>;
my $lucky_one = $pictures[rand(@pictures)];
die "Failed to pick a picture" unless $lucky_one;

print redirect("$PICTURE_URL/$lucky_one");

```

Under `mod_perl`, you would do this (the bigger size is because we're doing more checks here):

```

file:Apache/RandPicture.pm
-----
package Apache::RandPicture;

use strict;
use Apache::Constants qw(:common REDIRECT);
use DirHandle ();

sub handler {
    my $r = shift;
    my $dir_uri = $r->dir_config('PictureDir');

```

3.2.5 File Uploads

```
unless ($dir_uri) {
    $r->log_reason("No PictureDir configured");
    return SERVER_ERROR;
}
$dir_uri .= "/" unless $dir_uri =~ m:/$/;

my $subr = $r->lookup_uri($dir_uri);
my $dir = $subr->filename;
# Get list of images in the directory.
my $dh = DirHandle->new($dir);
unless ($dh) {
    $r->log_error("Can't read directory $dir: $!");
    return SERVER_ERROR;
}

my @files;
for my $entry ($dh->read) {
    # get the file's MIME type
    my $rr = $subr->lookup_uri($entry);
    my $type = $rr->content_type;
    next unless $type =~ m:^image/;;
    push @files, $rr->uri;
}
$dh->close;
unless (@files) {
    $r->log_error("No image files in directory");
    return SERVER_ERROR;
}

my $lucky_one = $files[rand @files];
# internal redirect, so we don't have to go back to the client
$r->internal_redirect($lucky_one);
return REDIRECT;
}

1;
__END__
```

3.2.5 File Uploads

Everyone wants to do it. I don't know why. Script I.5.1 shows a basic script that accepts a file to upload, reads it, and prints out its length and MIME type. Windows users should read about `binmode()` before they try this at home!

```
Script I.5.1 upload.pl
-----
#!/usr/local/bin/perl
#script: upload.pl

use CGI qw/:standard/;

print header,
    start_html('file upload'),
    h1('file upload');
print_form() unless param;
```

```

print_results() if param;
print end_html;

sub print_form {
    print start_multipart_form(),
        filefield(-name=>'upload',-size=>60),br,
        submit(-label=>'Upload File'),
        end_form;
}

sub print_results {
    my $length;
    my $file = param('upload');
    if (!$file) {
        print "No file uploaded.";
        return;
    }
    print h2('File name'),$file;
    print h2('File MIME type'),
        uploadInfo($file)->{'Content-Type'};
    while (<$file>) {
        $length += length($_);
    }
    print h2('File length'),$length;
}

```

3.3 Part II: Web Site Care and Feeding

These scripts are designed to make your life as a Webmaster easier, leaving you time for more exciting things, like tango lessons.

3.3.1 *Logs! Logs! Logs!*

Left to their own devices, the log files will grow without limit, eventually filling up your server's partition and bringing things to a grinding halt. But wait! Don't turn off logging or throw them away. Log files are your friends.

3.3.1.1 Log rotation

Script II.1.1 shows the basic script for rotating log files. It renames the current "access_log" to "access_log.0", "access_log.0" to "access_log.1", and so on. The oldest log gets deleted. Run it from a cron job to keep your log files from taking over. The faster your log files grow, the more frequently you should run the script.

```

Script II.1.1: Basic Log File Rotation
-----
#!/usr/local/bin/perl
$LOGPATH='/usr/local/apache/logs';
@LOGNAMES=('access_log','error_log','referer_log','agent_log');
$PIDFILE = 'httpd.pid';
$MAXCYCLE = 4;

```

3.3.1 Logs! Logs! Logs!

```
chdir $LOGPATH; # Change to the log directory
foreach $filename (@LOGNAMES) {
    for (my $s=$MAXCYCLE; $s >= 0; $s-- ) {
        $oldname = $s ? "$filename.$s" : $filename;
        $newname = join(".", $filename, $s+1);
        rename $oldname, $newname if -e $oldname;
    }
}
kill 'HUP', `cat $PIDFILE`;
```

3.3.1.2 Log rotation and archiving

But some people don't want to delete the old logs. Wow, maybe some day you could sell them for a lot of money to a marketing and merchandising company! Script II.1.2 appends the oldest to a gzip archive. Log files compress extremely well and make great bedtime reading.

```
Script II.1.2: Log File Rotation and Archiving
-----
#!/usr/local/bin/perl
$LOGPATH      = '/usr/local/apache/logs';
$PIDFILE      = 'httpd.pid';
$MAXCYCLE     = 4;
$GZIP         = '/bin/gzip';

@LOGNAMES=( 'access_log', 'error_log', 'referer_log', 'agent_log' );
%ARCHIVE=( 'access_log'=>1, 'error_log'=>1 );

chdir $LOGPATH; # Change to the log directory
foreach $filename (@LOGNAMES) {
    system "$GZIP -c $filename.$MAXCYCLE >> $filename.gz"
        if -e "$filename.$MAXCYCLE" and $ARCHIVE{$filename};
    for (my $s=$MAXCYCLE; $s >= 0; $s-- ) {
        $oldname = $s ? "$filename.$s" : $filename;
        $newname = join(".", $filename, $s+1);
        rename $oldname, $newname if -e $oldname;
    }
}
kill 'HUP', `cat $PIDFILE`;
```

3.3.1.3 Log rotation, compression and archiving

What's that? Someone broke into your computer, stole your log files and now **he's** selling it to a Web marketing and merchandising company? Shame on them. And on you for letting it happen. Script II.1.3 uses *idea* (part of the SSLEay package) to encrypt the log before compressing it. You need GNU tar to run this one. The log files are individually compressed and encrypted, and stamped with the current date.

```
Script II.1.3: Log File Rotation and Encryption
-----
#!/usr/local/bin/perl
use POSIX 'strftime';

$LOGPATH      = '/home/www/logs';
$PIDFILE      = 'httpd.pid';
$MAXCYCLE     = 4;
```

```

$IDEA      = '/usr/local/ssl/bin/idea';
$GZIP      = '/bin/gzip';
$TAR       = '/bin/tar';
$PASSWDFILE = '/home/www/logs/secret.passwd';

@LOGNAMES=('access_log','error_log','referer_log','agent_log');
%ARCHIVE=('access_log'=>1,'error_log'=>1);

chdir $LOGPATH; # Change to the log directory
foreach $filename (@LOGNAMES) {
    my $oldest = "$filename.$MAXCYCLE";
    archive($oldest) if -e $oldest and $ARCHIVE{$filename};
    for (my $s=$MAXCYCLE; $s >= 0; $s-- ) {
        $oldname = $s ? "$filename.$s" : $filename;
        $newname = join(".", $filename, $s+1);
        rename $oldname, $newname if -e $oldname;
    }
}
kill 'HUP', `cat $PIDFILE`;

sub archive {
    my $f = shift;
    my $base = $f;
    $base =~ s/\.d+$//;
    my $fn = strftime("$base.%Y-%m-%d_%H:%M.gz.idea", localtime);
    system "$GZIP -9 -c $f | $IDEA -kfile $PASSWDFILE > $fn";
    system "$TAR rvf $base.tar --remove-files $fn";
}

```

3.3.1.4 Log Parsing

There's a lot you can learn from log files. Script II.1.4 does the basic access log regular expression match. What you do with the split-out fields is limited by your imagination. Here's a typical log entry so that you can follow along (wrapped for readability):

```

portio.cshl.org - - [03/Feb/1998:17:42:15 -0500]
"GET /pictures/small_logo.gif HTTP/1.0" 200 2172

```

Script II.1.4: Basic Log Parsing

```

#!/usr/local/bin/perl

$REGEX=/^(\S+) (\S+) (\S+) \[([^\]]+\)] "(\w+) (\S+).*" (\d+) (\S+)/;
while (<>) {
    ($host,$rfc931,$user,$date,$request,$URL,$status,$bytes) = m/$REGEX/o;
    &collect_some_statistics;
}
&print_some_statistics;

sub collect_some_statistics {
    # for you to fill in
}

sub print_some_statistics {
    # for you to fill in
}

```

3.3.1 Logs! Logs! Logs!

Script II.1.5 scans the log for certain status codes and prints out the top URLs or hosts that triggered them. It can be used to get quick-and-dirty usage statistics, to find broken links, or to detect certain types of breakin attempts. Use it like this:

```
% find_status.pl -t10 200 ~www/logs/access_log
```

TOP 10 URLS/HOSTS WITH STATUS CODE 200:

REQUESTS	URL/HOST
1845	/www/wilogo.gif
1597	/cgi-bin/contig/sts_by_name?database=release
1582	/WWW/faqs/www-security-faq.html
1263	/icons/caution.xbm
930	/
886	/ftp/pub/software/WWW/cgi_docs.html
773	/cgi-bin/contig/phys_map
713	/icons/dna.gif
686	/WWW/pics/small_awlogo.gif

Script II.1.5: Find frequent status codes

```
-----
#!/usr/local/bin/perl
# File: find_status.pl

require "getopts.pl";
&Getopts('L:t:h') || die <<USAGE;
Usage: find_status.pl [-Lth] <code1> <code2> <code3> ...
    Scan Web server log files and list a summary
    of URLs whose requests had the one of the
    indicated status codes.
Options:
    -L <domain>  Ignore local hosts matching this domain
    -t <integer> Print top integer URLs/HOSTS [10]
    -h          Sort by host rather than URL
USAGE
;
if ($opt_L) {
    $opt_L=~s/\.\./\./g;
    $IGNORE = "([^\.]|$opt_L)\$";
}
$TOP=$opt_t || 10;

while (@ARGV) {
    last unless $ARGV[0]=~/^\d+$/;
    $CODES{shift @ARGV}++;
}

while (<>) {
    ($host,$rfc931,$user,$date,$request,$URL,$status,$bytes) =
        /^(S+) (S+) (S+) \[[^\]]+\] "(\w+) (S+).*" (\d+) (S+)/;
    next unless $CODES{$status};
    next if $IGNORE && $host=~/$IGNORE/io;
    $info = $opt_h ? $host : $URL;
    $found{$status}->{$info}++;
}
```


3.3.1.5 Offline Reverse DNS Resolution

This script maintains a cache of resolved names. Because performance is more important than completeness, if an address doesn't resolve after two seconds, it moves on to the next one and never tries that name again.

57

3.3.1 Logs! Logs! Logs!

```
my @i = gethostbyaddr(pack('C4',split('.', $ip)),2);
alarm(0);
@i;
END
$CACHE{$ip} = $h[0];
return $CACHE{$ip} || $ip;
}
```

3.3.1.6 Detecting Robots

I was very upset a few months ago when I did some log analysis and discovered that 90% of my hits were coming from 10% of users, and that those 10% were all robots! Script II.1.7 is the script I used to crunch the log and perform the analysis. The script works like this:

1. we assume that anyone coming from the same IP address with the same user agent within 30 minutes is the same person/robot (not quite right, but close enough).
2. anything that fetches /robots.txt is probably a robot, and a "polite" one, to boot.
3. we count the total number of accesses a user agent makes.
4. we average the interval between successive fetches.
5. we calculate an "index" which is the number of hits over the interval. Robots have higher indexes than people.
6. we print everything out in a big tab-delimited table for graphing.

By comparing the distribution of "polite" robots to the total distribution, we can make a good guess as to who the impolite robots are.

```
Script II.1.7: Robo-Cop
-----
#!/usr/local/bin/perl

use Time::ParseDate;
use strict 'vars';

# after 30 minutes, we consider this a new session
use constant MAX_INTERVAL => 60*30;
my (%HITS,%INT_NUMERATOR,%INT_DENOMINATOR,%POLITE,%LAST,$HITS);

# This uses a non-standard agent log with lines formatted like this:
# [08/Feb/1998:12:28:35 -0500] phila249-pri.voicenet.com "Mozilla/3.01 (Win95; U)" /cgi-bin/fortune

my $file = shift;
open (IN,$file=~ /\.gz$/ ? "zcat $file |" : $file) || die "Can't open file/pipe: $!";

while (<IN>) {
    my($date,$host,$agent,$URL) = /^\[([.+]\\) (\S+) "(.*)" (\S+)\$/;
    next unless $URL=~ /\. (html|htm|txt)$/;

    $HITS++;
    $host = "$host:$agent"; # concatenate host and agent
    $HITS{$host}++;
    my $seconds = parsedate($date);
    if ($LAST{$host}) {
```

```

    my $interval = $seconds - $LAST{$host};
    if ($interval < MAX_INTERVAL) {
        $INT_NUMERATOR{$host} += $interval;
        $INT_DENOMINATOR{$host}++;
    }
}
$LAST{$host} = $seconds;
$POLITE{$host}++ if $URL eq '/robots.txt';
print STDERR $HITS, "\n" if ($HITS % 1000) == 0;
}

# print out, sorted by hits
print join("\t", qw/Client Robot Hits Interval Hit_Percent Index/), "\n";
foreach (sort {$HITS{$b} <=> $HITS{$a}} keys %HITS) {

    next unless $HITS{$_} >= 5;          # not enough total hits to mean much
    next unless $INT_DENOMINATOR{$_} >= 5; # not enough consecutive hits to mean much

    my $mean_interval = $INT_NUMERATOR{$_}/$INT_DENOMINATOR{$_};
    my $percent_hits = 100*($HITS{$_}/$HITS);
    my $index = $percent_hits/$mean_interval;

    print join("\t",
        $_,
        $POLITE{$_} ? 'yes' : 'no',
        $HITS{$_},
        $mean_interval,
        $percent_hits,
        $index
    ), "\n";
}

```

3.3.1.7 Logging to syslog

If you run a large site with many independent servers, you might be annoyed that they all log into their own file systems rather than into a central location. Apache offers a little-known feature that allows it to send its log entries to a process rather than a file. The process (a Perl script, `natch`) can do whatever it likes with the logs. For instance, using Tom Christiansen's `Syslog` module to send the info to a remote syslog daemon.

Here's what you add to the Apache `httpd.conf` file:

```

<VirtualHost www.company1.com>
    CustomLog "| /usr/local/apache/bin/logger company1" common
    # blah blah
</VirtualHost>

<VirtualHost www.company2.com>
    CustomLog "| /usr/local/apache/bin/logger company2" common
    # blah blah
</VirtualHost>

```

Do the same for each server on the local network.

3.3.1 Logs! Logs! Logs!

Here's what you add to each Web server's syslog.conf (this assumes that the central logging host has the alias hostname "loghost"):

```
local0.info                                @loghost
```

Here's what you add to the central log host's syslog.conf:

```
local0.info                                /var/log/web/access_log
```

Script II.1.8 shows the code for the "logger" program:

```
Script II.1.8 "logger"
-----
#!/usr/local/bin/perl
# script: logger

use Sys::Syslog;

$SERVER_NAME = shift || 'www';
$FACILITY = 'local0';
$PRIORITY = 'info';

Sys::Syslog::setlogsock('unix');
openlog ($SERVER_NAME, 'ndelay', $FACILITY);
while (<>) {
    chomp;
    syslog($PRIORITY, $_);
}
closelog;
```

3.3.1.8 Logging to a relational database

One of the selling points of the big commercial Web servers is that they can log to relational databases via ODBC. Big whoop. With a little help from Perl, Apache can do that too. Once you've got the log in a relational database, you can data mine to your heart's content.

This example uses the freeware MySQL DBMS. To prepare, create an appropriate database containing a table named "access_log". It should have a structure like this one. Add whatever indexes you think you need. Also notice that we truncate URLs at 255 characters. You might want to use TEXT columns instead.

```
CREATE TABLE access_log (
    when      datetime      not null,
    host      varchar(255) not null,
    method    char(4)       not null,
    url       varchar(255) not null,
    auth      varchar(50),
    browser   varchar(50),
    referer   varchar(255),
    status    smallint(3)   not null,
    bytes     int(8)        default 0
);
```

Now create the following entries in `httpd.conf`:

```
LogFormat "%Y-%m-%d %H:%M:%S"t\" %h \"%r\" %u \"%{User-agent}i\" %{Referer}i %s %b" mysql
CustomLog "| /usr/local/apache/bin/mysqllog" mysql
```

Script II.1.9 is the source code for `mysqllog`.

```
Script II.1.9 "mysqllog"
-----
#!/usr/local/bin/perl
# script: mysqllog
use DBI;

use constant DSN          => 'dbi:mysql:www';
use constant DB_TABLE     => 'access_log';
use constant DB_USER      => 'nobody';
use constant DB_PASSWD    => '';

$PATTERN = '("[^"]+)" (\S+) "(\S+) (\S+) "[^"]+" (\S+) "("[^"]+)" (\S+) (\d+) (\S+)';

$db = DBI->connect(DSN,DB_USER,DB_PASSWD) || die DBI->errstr;
$stmt = $db->prepare("INSERT INTO ${DB_TABLE} VALUES(?,?,?,?,?,?,?,?,?)")
|| die $db->errstr;
while (<>) {
    chomp;
    my($date,$host,$method,$url,$user,$browser,$referer,$status,$bytes) = /$PATTERN/o;
    $user = undef if $user eq '-';
    $referer = undef if $referer eq '-';
    $browser = undef if $browser eq '-';
    $bytes = undef if $bytes eq '-';
    $stmt->execute($date,$host,$method,$url,$user,$browser,$referer,$status,$bytes);
}
$stmt->finish;
$db->disconnect;
```

NOTE: Your database will grow very quickly. Make sure that you have a plan for truncating or archiving the oldest entries. Or have a lot of storage space handy! Also be aware that this will cause a lot of traffic on your LAN. Better start shopping around for 100BT hubs.

3.3.2 *My server fell down and it can't get up!*

Web servers are very stable and will stay up for long periods of time if you don't mess with them. However, human error can bring them down, particularly if you have a lot of developers and authors involved in running the site. The scripts in this section watch the server and send you an email message when there's a problem.

3.3.2.1 Monitoring a local server

The simplest script just tries to signal the Web server process. If the process has gone away, it sends out an S.O.S. See script II.2.1 shows the technique. Notice that the script has to run as *root* in order to successfully signal the server.

3.3.2 My server fell down and it can't get up!

```
Script II.2.1 "localSOS"
-----
#!/usr/local/bin/perl
# script: localSOS

use constant PIDFILE => '/usr/local/apache/var/run/httpd.pid';
$MAIL                 = '/usr/sbin/sendmail';
$MAIL_FLAGS           = '-t -oi';
$WEBMASTER             = 'webmaster';

open (PID,PIDFILE) || die PIDFILE,": $!\n";
$pid = <PID>; close PID;
kill 0,$pid || sos();

sub sos {
    open (MAIL,"| $MAIL $MAIL_FLAGS") || die "mail: $!";
    my $date = localtime();
    print MAIL <<END;
    To: $WEBMASTER
    From: The Watchful Web Server Monitor <nobody>
    Subject: Web server is down

    I tried to call the Web server at $date but there was
    no answer.

    Respectfully yours,

    The Watchful Web Server Monitor
    END
    close MAIL;
}
```

3.3.2.2 Monitoring a remote server

Local monitoring won't catch problems with remote machines, and they'll miss subtle problems that can happen when the Web server hangs but doesn't actually crash. A functional test is better. Script II.2.2 uses the LWP library to send a HEAD request to a bunch of servers. If any of them fails to respond, it sends out an SOS. This script does **not** have to run as a privileged user.

```
Script II.2.2 "remoteSOS"
-----
#!/usr/local/bin/perl
# script: remoteSOS

use LWP::Simple;
%SERVERS = (
    "Fred's server"    => 'http://www.fred.com',
    "Martha's server" => 'http://www.stewart-living.com',
    "Bill's server"   => 'http://www.whitehouse.gov'
);

$MAIL                 = '/usr/sbin/sendmail';
$MAIL_FLAGS           = '-t -oi';
$WEBMASTER             = 'webmaster';

foreach (sort keys %SERVERS) {
    sos($_) unless head($SERVERS{$_});
}
```

```

}

sub sos {
    my $server = shift;
    open (MAIL,"| $MAIL $MAIL_FLAGS") || die "mail: $!";
    my $date = localtime();
    print MAIL <<END;
    To: $WEBMASTER
    From: The Watchful Web Server Monitor <nobody>
    Subject: $server is down

    I tried to call $server at $date but there was
    no one at home.

    Respectfully yours,

    The Watchful Web Server Monitor
    END
    close MAIL;
}

```

3.3.2.3 Resurrecting Dead Servers

So it's not enough to get e-mail that the server's down, you want to relaunch it as well? Script II.2.3 is a hybrid of localSOS and remoteSOS that tries to relaunch the local server after sending out the SOS. It has to be run as **root**, unless you've made *apachectl* *suid* to root.

```

Script II.2.2 "webLazarus"
-----
#!/usr/local/bin/perl
# script: webLazarus

use LWP::Simple;
use constant URL      => 'http://presto.capricorn.com/';
use constant APACHECTL => '/usr/local/apache/bin/apachectl';
$MAIL               = '/usr/sbin/sendmail';
$MAIL_FLAGS         = '-t -oi';
$WEBMASTER          = 'lstein@prego.capricorn.com';

head(URL) || resurrect();

sub resurrect {
    open (STDOUT,"| $MAIL $MAIL_FLAGS") || die "mail: $!";
    select STDOUT; $| = 1;
    open (STDERR,">&STDOUT");

    my $date = localtime();
    print <<END;
    To: $WEBMASTER
    From: The Watchful Web Server Monitor <nobody>
    Subject: Web server is down

    I tried to call the Web server at $date but there was
    no answer.  I am going to try to resurrect it now:

    Mumble, mumble, mumble, shazzzzammmmm!

```

3.3.3 Site Replication and Mirroring

```
END
;

system APACHECTL, 'restart';

print <<END;
```

That's the best I could do. Hope it helped.

Worshipfully yours,

```
The Web Monitor
END
    close STDERR;
    close STDOUT;
}
```

Here's the message you get when the script is successful:

```
Date: Sat, 4 Jul 1998 14:55:38 -0400
To: lstein@prego.capricorn.com
Subject: Web server is down
```

I tried to call the Web server at Sat Jul 4 14:55:37 1998 but there was no answer. I am going to try to resurrect it now:

Mumble, mumble, mumble, shazzzzammmmm!

```
/usr/local/apache/bin/apachectl restart: httpd not running, trying to start
[Sat Jul 4 14:55:38 1998] [debug] mod_so.c(258): loaded module setenvif_module
[Sat Jul 4 14:55:38 1998] [debug] mod_so.c(258): loaded module unique_id_module
/usr/local/apache/bin/apachectl restart: httpd started
```

That's the best I could do. Hope it helped.

Worshipfully yours,

The Web Monitor

3.3.3 *Site Replication and Mirroring*

Often you will want to mirror a page or set of pages from another server, for example, to distribute the load amongst several replicate servers, or to keep a set of reference pages handy. The LWP library makes this easy.

3.3.3.1 Mirroring Single Pages

```
% ./MirrorOne.pl
cats.html: Not Modified
dogs.html: OK
gillie_fish.html: Not Modified
```



```

Script II.3.1 mirrorOne.pl
-----
#!/usr/local/bin/perl
# mirrorOne.pl

use LWP::Simple;
use HTTP::Status;

use constant DIRECTORY => '/local/web/price_lists';
%DOCUMENTS = (
    'dogs.html'   => 'http://www.pets.com/dogs/price_list.html',
    'cats.html'   => 'http://www.pets.com/cats/price_list.html',
    'gillie_fish.html' => 'http://aquaria.com/prices.html'
);
chdir DIRECTORY;
foreach (sort keys %DOCUMENTS) {
    my $status = mirror($DOCUMENTS{$_}, $_);
    warn "$_: ", status_message($status), "\n";
}

```

3.3.3.2 Mirroring a Document Tree

With a little more work, you can recursively mirror an entire set of linked pages. Script II.3.2 mirrors the requested document and all subdocuments, using the LWP `HTML::LinkExtor` module to extract all the HTML links.

```

Script II.3.2 mirrorTree.pl
-----
#!/usr/local/bin/perl

# File: mirrorTree.pl

use LWP::UserAgent;
use HTML::LinkExtor;
use URI::URL;
use File::Path;
use File::Basename;
%DONE = ();

my $URL = shift;

$UA = new LWP::UserAgent;
$PARSER = HTML::LinkExtor->new();
$TOP = $UA->request(HTTP::Request->new(HEAD => $URL));
$BASE = $TOP->base;

mirror(URI::URL->new($TOP->request->url));

sub mirror {
    my $url = shift;

    # get rid of query string "?" and fragments "#"
    my $path = $url->path;
    my $fixed_url = URI::URL->new ($url->scheme . '://' . $url->netloc . $path);

    # make the URL relative

```

3.3.3 Site Replication and Mirroring

```
my $rel = $fixed_url->rel($BASE);
$rel .= 'index.html' if $rel=~m!/$! || length($rel) == 0;

# skip it if we've already done it
return if $DONE{$rel}++;

# create the directory if it doesn't exist already
my $dir = dirname($rel);
mkpath([$dir]) unless -d $dir;

# mirror the document
my $doc = $UA->mirror($fixed_url,$rel);
print STDERR "$rel: ",$doc->message,"\n";
return if $doc->is_error;

# Follow HTML documents
return unless $rel=~ /\.html?$/i;
my $base = $doc->base;

# pull out the links and call us recursively
my @links = $PARSER->parse_file("$rel")->links;
my @hrefs = map { url($_->[2],$base)->abs } @links;

foreach (@hrefs) {
    next unless is_child($BASE,$_);
    mirror($_);
}

}

sub is_child {
    my ($base,$url) = @_;
    my $rel = $url->rel($base);
    return ($rel ne $url) && ($rel !~ m!^[./.!]);
}
```

3.3.3.3 Checking for Bad Links

A slight modification of this last script allows you to check an entire document hierarchy (your own or someone else's) for bad links. The script shown in II.3.3 traverses a document, and checks each of the http:, ftp: and gopher: links to see if there's a response at the other end. Links that point to sub-documents are fetched and traversed as before, so you can check your whole site in this way.

```
% find_bad_links http://prego/apache-1.2/
checking http://prego/apache-1.2/...
checking http://prego/apache-1.2/manual/...
checking http://prego/apache-1.2/manual/misc/footer.html...
checking http://prego/apache-1.2/manual/misc/header.html...
checking http://prego/apache-1.2/manual/misc/nopgp.html...
checking http://www.yahoo.com/Science/Mathematics/Security_and_Encryption/...
checking http://www.eff.org/pub/EFF/Policy/Crypto/...
checking http://www.quadralay.com/www/Crypt/Crypt.html...
checking http://www.law.indiana.edu/law/iclu.html...
checking http://bong.com/~brian...
checking http://prego/apache-1.2/manual/cgi_path.html...
checking http://www.ics.uci.edu/pub/ietf/http/...
```

```

.
.
.
BAD LINKS:
manual/misc/known_bugs.html : http://www.apache.org/dist/patches/apply_to_1.2b6/
manual/misc/fin_wait_2.html : http://www.freebsd.org/
manual/misc/fin_wait_2.html : http://www.ncr.com/
manual/misc/compat_notes.html : http://www.eit.com/
manual/misc/howto.html : http://www.zyzyva.com/robots/alert/
manual/misc/perf.html : http://www.software.hp.com/internet/perf/tuning.html
manual/misc/perf.html : http://www.qosina.com/~awm/apache/linux-tcp.html
manual/misc/perf.html : http://www.sun.com/sun-on-net/Sun.Internet.Solutions/performance/
manual/misc/perf.html : http://www.sun.com/solaris/products/siss/
manual/misc/nopgp.html : http://www.yahoo.com/Science/Mathematics/Security_and_Encryption/

152 documents checked
11 bad links

Script II.3.2 find_bad_links.pl
-----
#!/usr/local/bin/perl

# File: find_bad_links.pl

use LWP::UserAgent;
use HTML::LinkExor;
use URI::URL;
use WWW::RobotRules;

%CAN_HANDLE = ( 'http'=>1,
                'gopher'=>1,
                # 'ftp'=>1,    # timeout problems?
                );

%OUTCOME = ();
$CHECKED = $BAD = 0;
@BAD = ();

my $URL = shift;

$UA      = new LWP::UserAgent;
$PARSER  = HTML::LinkExor->new();
$TOP     = $UA->request(HTTP::Request->new(HEAD => $URL));
$BASE    = $TOP->base;

# handle robot rules
my $robots = URI::URL->new('robots.txt',$BASE->scheme.'://'.$BASE->netloc);
my $robots_text = $UA->request(HTTP::Request->new(GET=>$robots))->content;
$ROBOTRULES = WWW::RobotRules->new;
$ROBOTRULES->parse($robots->abs,$robots_text);

check_links(URI::URL->new($TOP->request->url));
if (@BAD) {
    print "\nBAD LINKS:\n";
    print join("\n",@BAD),"\n\n";
}
print "$CHECKED documents checked\n",scalar(@BAD)," bad links\n";

sub check_links {

```

3.3.4 Load balancing

```
my $url = shift;
my $fixed_url = $url;
$fixed_url =~ s/\#.+$/;/

return 1 unless $CAN_HANDLE{$url->scheme};

# check cached outcomes
return $OUTCOME{$fixed_url} if exists $OUTCOME{$fixed_url};

print STDERR "checking $fixed_url...\n";
$CHECKED++;

my $rel = $url->rel($BASE) || 'index.html';
my $child = is_child($BASE,$url);
$UA->timeout(5);
my $doc = $d = $UA->request(HTTP::Request->new(($child ? 'GET' : 'HEAD' )=>$url));
$OUTCOME{$fixed_url} = $doc->is_success;

return $OUTCOME{$fixed_url}
unless $ROBOTRULES->allowed($fixed_url)
    && $child && $doc->header('Content-type') eq 'text/html';

# Follow HTML documents
my $base = $doc->base;

# pull out the links and call us recursively
my @links = $PARSER->parse($doc->content)->links;
my @hrefs = map { url($_->[2],$base)->abs } @links;

foreach (@hrefs) {
    next if check_links($_);
    push (@BAD,"$rel : $_");
}
1;
}

sub is_child {
my ($base,$url) = @_;
my $rel = $url->rel($base);
return ($rel ne $url) && ($rel !~ m![/.]!);
}
```

3.3.4 Load balancing

You've hit the big time, and your site is getting more hits than you ever dreamed of. Millions, zillions of hits. What's that? System load just passed 50 and response time is getting kinda' s-l-o-w-w-w?

Perl to the rescue. Set up several replica Web servers with different hostnames and IP addresses. Run this script on the "main" site and watch it round-robin the requests to the replica servers. It uses `IO::Socket` to listen for incoming requests on port 80. It then changes its privileges to run as `nobody.nogroup`, just like a real Web server. Next it preforks itself a few times (and you always thought preforking was something fancy, didn't you?), and goes into an `accept()` loop. Each time an incoming session comes in, it forks off another child to handle the request. The child reads the HTTP request and issues the an HTTP redirec-

tion to send the browser to a randomly selected server.

NOTE: Another way to do this is to have multiple "A" records defined for your server's hostname and let DNS caching distribute the load.

```
Script II.4.1: A Load Balancing "Web Server"
-----
#!/usr/local/bin/perl

# list of hosts to balance between
@HOSTS = qw/www1.web.org www2.web.org www3.web.org www4.web.org/;

use IO::Socket;
$SIG{CHLD} = sub { wait() };
$ENV{'PATH'} = '/bin:/usr/bin';
chomp($hostname = `/bin/hostname`);

# Listen on port 80
$sock = IO::Socket::INET->new(Listen => 5,
                               LocalPort => 80,
                               LocalAddr => $hostname,
                               Reuse => 1,
                               Proto => 'tcp');

# become "nobody"
$nobody = (getpwnam('nobody'))[2] || die "nobody is nobody";
$nogroup = (getgrnam('nogroup'))[2] || die "can't grok nogroup";
($<,$()) = ($>,$) = ($nobody,$nogroup); # get rid of root privileges!
($\,$/) = ("\r\n","\r\n\r\n");          # CR/LF on output/input

# Go into server mode
close STDIN; close STDOUT; close STDERR;

# prefork -- gee is that all there is to it?
fork() && fork() && fork() && fork() && exit 0;

# start accepting connections
while (my $s = $sock->accept()) {
    do { $s->close; next; } if fork();
    my $request = <$s>;
    redirect($1,$s) if $request =~ /(?:GET|POST|HEAD|PUT)\s+(\S+)/;
    $s->flush;
    undef $s;
    exit 0;
}

sub redirect {
    my ($url,$s) = @_;
    my $host = $HOSTS[rand(@HOSTS)];
    print $s "HTTP/1.0 301 Moved Temporarily";
    print $s "Server: Lincoln's Redirector/1.0";
    print $s "Location: http://${host}${url}";
    print $s "";
}
```

3.3.5 Torture Testing a Server

Any server written in C suffers the risk of static buffer overflow bugs. In the past, these bugs have led to security compromises and Web server breakins. Script II.2.3 torture tests servers and CGI scripts by sending large amounts of random data to them. If the server crashes, it probably contains a buffer overflow bug.

Here's what you see when a server crashes:

```
% torture.pl -t 1000 -l 5000 http://www.capricorn.com
torture.pl version 1.0 starting
Base URL:          http://www.capricorn.com/cgi-bin/search
Max random data length: 5000
Repetitions:       1000
Post:              0
Append to path:    0
Escape URLs:       0

200 OK
200 OK
200 OK
200 OK
200 OK
500 Internal Server Error
500 Could not connect to www.capricorn.com:80
500 Could not connect to www.capricorn.com:80
500 Could not connect to www.capricorn.com:80

Script II.5.1: torture tester
-----
#!/usr/local/bin/perl

# file: torture.pl
# Torture test Web servers and scripts by sending them large arbitrary URLs
# and record the outcome.

use LWP::UserAgent;
use URI::Escape 'uri_escape';
require "getopts.pl";

$USAGE = <<USAGE;
Usage: $0 -[options] URL
Torture-test Web servers and CGI scripts

Options:
-l <integer>  Max length of random URL to send [1024 bytes]
-t <integer>  Number of times to run the test [1]
-P           Use POST method rather than GET method
-p          Attach random data to path rather than query string
-e          Escape the query string before sending it
USAGE

$VERSION = '1.0';

# process command line
```

```

&Getopts('l:t:Ppe') || die $USAGE;

# get parameters
$URL    = shift || die $USAGE;
$MAXLEN = $opt_l ne '' ? $opt_l : 1024;
$TIMES  = $opt_t || 1;
$POST   = $opt_P || 0;
$PATH   = $opt_p || 0;
$ESCAPE = $opt_e || 0;

# cannot do both a post and a path at the same time
$POST = 0 if $PATH;

# create an LWP agent
my $agent = new LWP::UserAgent;

print <<EOF;
torture.pl version $VERSION starting
Base URL:           $URL
Max random data length: $MAXLEN
Repetitions:        $TIMES
Post:               $POST
Append to path:      $PATH
Escape URLs:         $ESCAPE

EOF

# Do the test $TIMES times
while ($TIMES) {
    # create a string of random stuff
    my $garbage = random_string(rand($MAXLEN));
    $garbage = uri_escape($garbage) if $ESCAPE;
    my $url = $URL;
    my $request;

    if (length($garbage) == 0) { # if no garbage to add, just fetch URL
        $request = new HTTP::Request ('GET', $url);
    }

    elsif ($POST) { # handle POST request
        my $header = new HTTP::Headers (
            Content_Type => 'application/x-www-form-urlencoded',
            Content_Length => length($garbage)
        );
        # garbage becomes the POST content
        $request = new HTTP::Request ('POST', $url, $header, $garbage);
    } else { # handle GET request

        if ($PATH) { # append garbage to the base URL
            chop($url) if substr($url, -1, 1) eq '/';
            $url .= "/$garbage";
        } else { # append garbage to the query string
            $url .= "?$garbage";
        }

        $request = new HTTP::Request ('GET', $url);
    }
}

```

```

    }

    # do the request and fetch the response
    my $response = $agent->request($request);

    # print the numeric response code and the message
    print $response->code, ' ', $response->message, "\n";

} continue { $TIMES-- }

# return some random data of the requested length
sub random_string {
    my $length = shift;
    return undef unless $length >= 1;
    return join('', map chr(rand(255)), 0..$length-1);
}

```

For other load testing tools, have a look at our Benchmarking section.

3.4 Part III: mod_perl -- Faster Than a Speeding Bullet

mod_perl is Doug MacEachern's embedded Perl for Apache. With a *mod_perl*-enabled server, there's no tedious waiting around while the Perl interpreter fires up, reads and compiles your script. It's right there, ready and waiting. What's more, once compiled your script remains in memory, all charged and raring to go. Suddenly those sluggish Perl CGI scripts race along at compiled C speeds...or so it seems.

Most CGI scripts will run unmodified under *mod_perl* using the `Apache::Registry` CGI compatibility layer. But that's not the whole story. The exciting part is that *mod_perl* gives you access to the Apache API, letting you get at the innards of the Apache server and change its behavior in powerful and interesting ways. This section will give you a feel for the many things that you can do with *mod_perl*.

3.4.1 Creating Dynamic Pages

This is a ho-hum because you can do it with CGI and with `Apache::Registry`. Still, it's worth seeing a simple script written using the strict *mod_perl* API so you see what it looks like. Script III.1.1 prints out a little hello world message.

Install it by adding a section like this one to one of the configuration files:

```

<Location /hello/world>
    SetHandler perl-script
    PerlHandler Apache::Hello
</Location>

Script III.1.1 Apache::Hello
-----
package Apache::Hello;
# file: Apache/Hello.pm

use strict vars;
use Apache::Constants ':common';

```



```

sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->send_http_header;
    my $host = $r->get_remote_host;
    $r->print(<<END);
    <html>
    <head>
    <title>Hello There</title>
    </head>
    <body>
    <h1>Hello $host</h1>
    Hello to all the nice people at the Perl conference.  Lincoln is
    trying really hard.  Be kind.
    </body>
    </html>
    END
    return OK;
}
1;

```

You can do all the standard CGI stuff, such as reading the query string, creating fill-out forms, and so on. In fact, CGI.pm works with mod_perl, giving you the benefit of sticky forms, cookie handling, and elegant HTML generation.

3.4.2 File Filters

This is where the going gets fun. With mod_perl, you can install a *content handler* that works a lot like a four-letter word starrer-outer, but a lot faster.

3.4.2.1 Adding a Canned Footer to Every Page

Script III.2.1 adds a canned footer to every HTML file. The footer contains a copyright statement, plus the modification date of the file. You could easily extend this to add other information, such as a page hit counter, or the username of the page's owner.

This can be installed as the default handler for all files in a particular subdirectory like this:

```

<Location /footer>
    SetHandler perl-script
    PerlHandler Apache::Footer
</Location>

```

Or you can declare a new ".footer" extension and arrange for all files with this extension to be passed through the footer module:

```

AddType text/html .footer
<Files ~ "\.footer$">
    SetHandler perl-script
    PerlHandler Apache::Footer
</Files>

```

```

Script III.2.1 Apache::Footer
-----
package Apache::Footer;
# file Apache::Footer.pm

use strict vars;
use Apache::Constants ':common';
use IO::File;

sub handler {
    my $r = shift;
    return DECLINED unless $r->content_type() eq 'text/html';
    my $file = $r->filename;
    return DECLINED unless $fh=IO::File->new($file);
    my $mtime = localtime((stat($file))[9]);
    my $footer=<<END;
<hr>
&copy; 1998 <a href="http://www.ora.com/">O\ 'Reilly & Associates</a><br>
<em>Last Modified: $mtime</em>
END

    $r->send_http_header;

    while (<$fh>) {
        s!(</BODY>)!$footer$1!oi;
    } continue {
        $r->print($_);
    }

    return OK;
}

1;

```

For more customized footer/header handling, you might want to look at the `Apache::Sandwich` module on CPAN.

3.4.2.2 Dynamic Navigation Bar

Sick of hand-coding navigation bars in every HTML page? Less than enthused by the Java & JavaScript hacks? Here's a dynamic navigation bar implemented as a server side include.

First create a global configuration file for your site. The first column is the top of each major section. The second column is the label to print in the navigation bar

```

# Configuration file for the navigation bar
/index.html      Home
/new/            What's New
/tech/           Tech Support
/download/       Download
/dev/zero        Customer support
/dev/null        Complaints

```

Then, at the top (or bottom) of each HTML page that you want the navigation bar to appear on, add this comment:

```
<!--#NAVBAR-->
```

Now add `Apache::NavBar` to your system (Script III.2.2). This module parses the configuration file to create a "navigation bar object". We then call the navigation bar object's `to_html()` method in order to generate the HTML for the navigation bar to display on the current page (it will be different for each page, depending on what major section the page is in).

The next section does some checking to avoid transmitting the page again if it is already cached on the browser. The effective last modified time for the page is either the modification time of its HTML source code, or the navbar's configuration file modification date, whichever is more recent.

The remainder is just looping through the file a section at a time, searching for the `<!--NAVBAR-->` comment, and substituting the navigation bar HTML.

```
Script III.2.2 Apache::NavBar
-----

package Apache::NavBar;
# file Apache/NavBar.pm

use strict;
use Apache::Constants qw(:common);
use Apache::File ();

my %BARS = ();
my $TABLEATTRS = 'WIDTH="100%" BORDER=1';
my $TABLECOLOR = '#C8FFFF';
my $ACTIVECOLOR = '#FF0000';

sub handler {
    my $r = shift;

    my $bar = read_configuration($r) || return DECLINED;
    $r->content_type eq 'text/html' || return DECLINED;
    my $fh = Apache::File->new($r->filename) || return DECLINED;
    my $navbar = $bar->to_html($r->uri);

    $r->update_mtime($bar->modified);
    $r->set_last_modified;
    my $rc = $r->meets_conditions;
    return $rc unless $rc == OK;

    $r->send_http_header;
    return OK if $r->header_only;

    local $/ = "";
    while (<$fh>) {
        s:<!--NAVBAR-->:$navbar:oi;
    } continue {
        $r->print($_);
    }
}
```

3.4.2 File Filters

```
    }

    return OK;
}

# read the navigation bar configuration file and return it as a
# hash.
sub read_configuration {
    my $r = shift;
    my $conf_file;
    return unless $conf_file = $r->dir_config('NavConf');
    return unless -e ($conf_file = $r->server_root_relative($conf_file));
    my $mod_time = (stat _)[9];
    return $BARS{$conf_file} if $BARS{$conf_file}
        && $BARS{$conf_file}->modified >= $mod_time;
    return $BARS{$conf_file} = NavBar->new($conf_file);
}

package NavBar;

# create a new NavBar object
sub new {
    my ($class,$conf_file) = @_ ;
    my (@c,%c);
    my $fh = Apache::File->new($conf_file) || return;
    while (<$fh>) {
        chomp;
        s/^\s+//; s/\s+$//; #fold leading and trailing whitespace
        next if /^#/ || /^$/; # skip comments and empty lines
        next unless my($url, $label) = /^(\S+)\s+(.+)$/;
        push @c, $url; # keep the url in an ordered array
        $c{$url} = $label; # keep its label in a hash
    }
    return bless {'urls' => \@c,
                  'labels' => \%c,
                  'modified' => (stat $conf_file)[9]}, $class;
}

# return ordered list of all the URIs in the navigation bar
sub urls { return @{$shift->{'urls'}}; }

# return the label for a particular URI in the navigation bar
sub label { return $_[0]->{'labels'}->{$_[1]} || $_[1]; }

# return the modification date of the configuration file
sub modified { return $_[0]->{'modified'}; }

sub to_html {
    my $self = shift;
    my $current_url = shift;
    my @cells;
    for my $url ($self->urls) {
        my $label = $self->label($url);
        my $is_current = $current_url =~ /^$url/;
        my $cell = $is_current ?
            qq(<FONT COLOR="$ACTIVECOLOR">$label</FONT>)
    }
}
```

```

        : qq(<A HREF="$url">$label</A>);
    push @cells,
    qq(<TD CLASS="navbar" ALIGN=CENTER BGCOLOR="$TABLECOLOR">$cell</TD>\n);
}
return qq(<TABLE $TABLEATTRS><TR>@cells</TR></TABLE>\n);
}

1;
__END__

<Location />
    SetHandler perl-script
    PerlHandler Apache::NavBar
    PerlSetVar NavConf etc/navigation.conf
</Location>

```

Apache::NavBar is available on the CPAN, with further improvements.

3.4.2.3 On-the-Fly Compression

WU-FTP has a great feature that automatically gzips a file if you fetch it by name with a .gz extension added. Why can't Web servers do that trick? With Apache and mod_perl, you can.

Script III.2.4 is a content filter that automatically gzips everything retrieved from a particular directory and adds the "gzip" Content-Encoding header to it. Unix versions of Netscape Navigator will automatically recognize this encoding type and decompress the file on the fly. Windows and Mac versions don't. You'll have to save to disk and decompress, or install the WinZip plug-in. Bummer.

The code uses the Compress::Zlib module, and has to do a little fancy footwork (but not too much) to create the correct gzip header. You can extend this idea to do on-the-fly encryption, or whatever you like.

Here's the configuration entry you'll need. Everything in the */compressed* directory will be compressed automatically.

```

<Location /compressed>
    SetHandler perl-script
    PerlHandler Apache::GZip
</Location>

Script III.2.3: Apache::GZip
-----
package Apache::GZip;
#File: Apache::GZip.pm

use strict vars;
use Apache::Constants ':common';
use Compress::Zlib;
use IO::File;
use constant GZIP_MAGIC => 0x1f8b;
use constant OS_MAGIC => 0x03;

sub handler {
    my $r = shift;

```

3.4.2 File Filters

```
my ($fh,$gz);
my $file = $r->filename;
return DECLINED unless $fh=IO::File->new($file);
$r->header_out('Content-Encoding'=>'gzip');
$r->send_http_header;
return OK if $r->header_only;

tie *STDOUT,'Apache::GZip',$r;
print($_) while <$fh>;
untie *STDOUT;
return OK;
}

sub TIEHANDLE {
    my($class,$r) = @_;
    # initialize a deflation stream
    my $d = deflateInit(-WindowBits=>-MAX_WBITS()) || return undef;

    # gzip header -- don't ask how I found out
    $r->print(pack("nccVcc",GZIP_MAGIC,Z_DEFLATED,0,time(),0,OS_MAGIC));

    return bless { r => $r,
                   crc => crc32(undef),
                   d => $d,
                   l => 0
                 },$class;
}

sub PRINT {
    my $self = shift;
    foreach (@_) {
        # deflate the data
        my $data = $self->{d}->deflate($_);
        $self->{r}->print($data);
        # keep track of its length and crc
        $self->{l} += length($_);
        $self->{crc} = crc32($_,$self->{crc});
    }
}

sub DESTROY {
    my $self = shift;

    # flush the output buffers
    my $data = $self->{d}->flush;
    $self->{r}->print($data);

    # print the CRC and the total length (uncompressed)
    $self->{r}->print(pack("LL",@{$self}{qw/crc l/}));
}

1;
```

For some alternatives that are being maintained, you might want to look at the `Apache::Compress` and `Apache::GzipChain` modules on CPAN, which can handle the output of any handler in a chain.

By adding a URI translation handler, you can set things up so that a remote user can append a `.gz` to the end of any URL and the file we be delivered in compressed form. Script III.2.4 shows the translation handler you need. It is called during the initial phases of the request to make any modifications to the URL that it wishes. In this case, it removes the `.gz` ending from the filename and arranges for `Apache::GZip` to be called as the content handler. The `lookup_uri()` call is used to exclude anything that has a special handler already defined (such as CGI scripts), and actual gzip files. The module replaces the information in the request object with information about the real file (without the `.gz`), and arranges for `Apache::GZip` to be the content handler for this file.

You just need this one directive to activate handling for all URLs at your site:

```
PerlTransHandler Apache::AutoGZip

Script III.2.4: Apache::AutoGZip
-----
package Apache::AutoGZip;

use strict 'vars';
use Apache::Constants qw/:common/;

sub handler {
    my $r = shift;

    # don't allow ourselves to be called recursively
    return DECLINED unless $r->is_initial_req;

    # don't do anything for files not ending with .gz
    my $uri = $r->uri;
    return DECLINED unless $uri =~ /\.gz$/;
    my $basename = $';

    # don't do anything special if the file actually exists
    return DECLINED if -e $r->lookup_uri($uri)->filename;

    # look up information about the file
    my $subr = $r->lookup_uri($basename);
    $r->uri($basename);
    $r->path_info($subr->path_info);
    $r->filename($subr->filename);

    # fix the handler to point to Apache::GZip;
    my $handler = $subr->handler;
    unless ($handler) {
        $r->handler('perl-script');
        $r->push_handlers('PerlHandler', 'Apache::GZip');
    } else {
        $r->handler($handler);
    }
}
```

```

        return OK;
    }

    1;

```

3.4.3 Access Control

Access control, as opposed to authentication and authorization, is based on something the user "is" rather than something he "knows". The "is" is usually something about his browser, such as its IP address, host-name, or user agent. Script III.3.1 blocks access to the Web server for certain User Agents (you might use this to block impolite robots).

Apache::BlockAgent reads its blocking information from a "bad agents" file, which contains a series of pattern matches. Most of the complexity of the code comes from watching this file and recompiling it when it changes. If the file doesn't change, it's only read once and its patterns compiled in memory, making this module fast.

Here's an example bad agents file:

```

^teleport pro\1\.28
^nicerspro
^mozilla\3\.0 \ (http engine\ )
^netattache
^crescent internet toolpak http ole control v\1\.0
^go-ahead-got-it
^wget
^devsoft's http component v1\.0
^www\.pl
^digout4uagent

```

A configuration entry to activate this blocker looks like this. In this case we're blocking access to the entire site. You could also block access to a portion of the site, or have different bad agents files associated with different portions of the document tree.

```

<Location />
    PerlAccessHandler Apache::BlockAgent
    PerlSetVar BlockAgentFile /home/www/conf/bad_agents.txt
</Location>

```

```

Script III.3.1: Apache::BlockAgent
-----
package Apache::BlockAgent;
# block browsers that we don't like

use strict 'vars';
use Apache::Constants ':common';
use IO::File;
my %MATCH_CACHE;
my $DEBUG = 0;

sub handler {
    my $r = shift;

```



```

    return DECLINED unless my $patfile = $r->dir_config('BlockAgentFile');
    return FORBIDDEN unless my $agent = $r->header_in('User-Agent');
    return SERVER_ERROR unless my $sub = get_match_sub($r,$patfile);
    return OK if $sub->($agent);
    $r->log_reason("Access forbidden to agent $agent",$r->filename);
    return FORBIDDEN;
}

# This routine creates a pattern matching subroutine from a
# list of pattern matches stored in a file.
sub get_match_sub {
    my ($r,$filename) = @_ ;
    my $mtime = -M $filename;

    # try to return the sub from cache
    return $MATCH_CACHE{$filename}->{'sub'} if
        $MATCH_CACHE{$filename} &&
        $MATCH_CACHE{$filename}->{'mod'} <= $mtime;

    # if we get here, then we need to create the sub
    return undef unless my $fh = new IO::File($filename);
    chomp(my @pats = <$fh>); # get the patterns into an array
    my $code = "sub { \$_ = shift;\n";
    foreach (@pats) {
        next if /^#/
        $code .= "return undef if /\$_/i;\n";
    }
    $code .= "1; }\n";
    warn $code if $DEBUG;

    # create the sub, cache and return it
    my $sub = eval $code;
    unless ($sub) {
        $r->log_error($r->uri," : ",$@);
        return undef;
    }
    @{$MATCH_CACHE{$filename}}{'sub','mod'}=($sub,$mtime);
    return $MATCH_CACHE{$filename}->{'sub'};
}

1;

```

3.4.4 Authentication and Authorization

Thought you were stuck with authentication using text, DBI and DBM files? `mod_perl` opens the authentication/authorization API wide. The two phases are authentication, in which the user has to prove who he or she is (usually by providing a username and password), and authorization, in which the system decides whether this user has sufficient privileges to view the requested URL. A scheme can incorporate authentication and authorization either together or singly.

3.4.4.1 Authentication with NIS

If you keep Unix system passwords in */etc/passwd* or distribute them by NIS (not NIS+) you can authenticate Web users against the system password database. (It's not a good idea to do this if the system is connected to the Internet because passwords travel in the clear, but it's OK for trusted intranets.)

Script III.4.1 shows how the `Apache::AuthSystem` module fetches the user's name and password, compares it to the system password, and takes appropriate action. The `getpwnam()` function operates either on local files or on the NIS database, depending on how the server host is configured. **WARNING:** the module will fail if you use a shadow password system, since the Web server doesn't have root privileges.

In order to activate this system, put a configuration directive like this one in *access.conf*:

```
<Location /protected>
    AuthName Test
    AuthType Basic
    PerlAuthenHandler Apache::AuthSystem
    require valid-user
</Location>

Script III.4.1: Apache::AuthSystem
-----
package Apache::AuthSystem;
# authenticate users on system password database

use strict;
use Apache::Constants ':common';

sub handler {
    my $r = shift;

    my($res, $sent_pwd) = $r->get_basic_auth_pwd;
    return $res if $res != OK;

    my $user = $r->connection->user;
    my $reason = "";

    my($name,$passwd) = getpwnam($user);
    if (!$name) {
        $reason = "user does not have an account on this system";
    } else {
        $reason = "user did not provide correct password"
            unless $passwd eq crypt($sent_pwd,$passwd);
    }

    if($reason) {
        $r->note_basic_auth_failure;
        $r->log_reason($reason,$r->filename);
        return AUTH_REQUIRED;
    }
}
```

```

    return OK;
}

1;

```

There are modules doing equivalent things on CPAN: `Apache::AuthenPasswd` and `Apache::AuthxPasswd`.

3.4.4.2 Anonymous Authentication

Here's a system that authenticates users the way anonymous FTP does. They have to enter a name like "Anonymous" (configurable) and a password that looks like a valid e-mail address. The system rejects the username and password unless they are formatted correctly.

In a real application, you'd probably want to log the password somewhere for posterity. Script III.4.2 shows the code for `Apache::AuthAnon`. To activate it, create a *httpd.conf* section like this one:

```

<Location /protected>
    AuthName Anonymous
    AuthType Basic
    PerlAuthenHandler Apache::AuthAnon
    require valid-user

    PerlSetVar Anonymous anonymous|anybody
</Location>

Script III.4.2: Anonymous Authentication
-----
package Apache::AuthAnon;

use strict;
use Apache::Constants ':common';

my $email_pat = '\w+@\w+\.\w+';
my $anon_id = "anonymous";

sub handler {
    my $r = shift;

    my($res, $sent_pwd) = $r->get_basic_auth_pw;
    return $res if $res != OK;

    my $user = lc $r->connection->user;
    my $reason = "";

    my $check_id = $r->dir_config("Anonymous") || $anon_id;

    unless($user =~ /^$check_id$/i) {
        $reason = "user did not enter a valid anonymous username";
    }

    unless($sent_pwd =~ /$email_pat/o) {
        $reason = "user did not enter an email address password";
    }
}

```

```

    if($reason) {
        $r->note_basic_auth_failure;
        $r->log_reason($reason,$r->filename);
        return AUTH_REQUIRED;
    }

    $r->notes(AuthAnonPassword => $sent_pwd);

    return OK;
}

1;

```

3.4.4.3 Gender-Based Authorization

After authenticating, you can authorize. The most familiar type of authorization checks a group database to see if the user belongs to one or more privileged groups. But authorization can be anything you dream up.

Script III.4.3 shows how you can authorize users by their gender (or at least their *apparent* gender, by checking their names with Jon Orwant's `Text::GenderFromName` module. This must be used in conjunction with an authentication module, such as one of the standard Apache modules or a custom one.

This configuration restricts access to users with feminine names, except for the users "Webmaster" and "Jeff", who are allowed access.

```

<Location /ladies_only>
    AuthName "Ladies Only"
    AuthType Basic
    AuthUserFile /home/www/conf/users.passwd
    PerlAuthzHandler Apache::AuthzGender
    require gender F # allow females
    require user Webmaster Jeff # allow Webmaster or Jeff
</Location>

```

The script uses a custom error response to explain why the user was denied admittance. This is better than the standard "Authorization Failed" message.

```

Script III.4.3: Apache::AuthzGender
-----
package Apache::AuthzGender;

use strict;
use Text::GenderFromName;
use Apache::Constants ":common";

my %G=( 'M'=>"male", 'F'=>"female");

sub handler {
    my $r = shift;

    return DECLINED unless my $requires = $r->requires;
    my $user = lc($r->connection->user);
    substr($user,0,1)=~tr/a-z/A-Z/;

```

```

my $guessed_gender = uc(gender($user)) || 'M';

my $explanation = <<END;
<html><head><title>Unauthorized</title></head><body>
<h1>You Are Not Authorized to Access This Page</h1>
Access to this page is limited to:
<ol>
END

foreach (@$requires) {
    my ($requirement,@rest) = split(/\s+/, $_->{requirement});
    if (lc $requirement eq 'user') {
        foreach (@rest) { return OK if $user eq $_; }
        $explanation .= "<LI>Users @rest.\n";
    } elsif (lc $requirement eq 'gender') {
        foreach (@rest) { return OK if $guessed_gender eq uc $_; }
        $explanation .= "<LI>People of the @G{@rest} persuasion.\n";
    } elsif (lc $requirement eq 'valid-user') {
        return OK;
    }
}

$explanation .= "</OL></BODY></HTML>";

$r->custom_response(AUTH_REQUIRED,$explanation);
$r->note_basic_auth_failure;
$r->log_reason("user $user: not authorized",$r->filename);
return AUTH_REQUIRED;
}

1;

```

Apache::AuthzGender is available from the CPAN.

3.4.5 Proxy Services

mod_perl gives you access to Apache's ability to act as a Web proxy. You can intervene at any step in the proxy transaction to modify the outgoing request (for example, stripping off headers in order to create an anonymizing proxy) or to modify the returned page.

3.4.5.1 A Banner Ad Blocker

Script III.5.1 shows the code for a banner-ad blocker written by Doug MacEachern. It intercepts all proxy requests, substituting its own content handler for the default. The content handler uses the LWP library to fetch the requested document. If the retrieved document is an image, and its URL matches the pattern (ads?[advertisement|banner]), then the content of the image is replaced with a dynamically-generated GIF that reads "Blocked Ad". The generated image is exactly the same size as the original, preserving the page layout. Notice how the outgoing headers from the Apache request object are copied to the LWP request, and how the incoming LWP response headers are copied back to Apache. This makes the transaction nearly transparent to Apache and to the remote server.

In addition to LWP you'll need `GD.pm` and `Image::Size` to run this module. To activate it, add the following line to the configuration file:

```
PerlTransHandler Apache::AdBlocker
```

Then configure your browser to use the server to proxy all its HTTP requests. Works like a charm! With a little more work, and some help from the `ImageMagick` module, you could adapt this module to quiet-down animated GIFs by stripping them of all but the very first frame.

```
Script III.5.1: Apache::AdBlocker
-----

package Apache::AdBlocker;

use strict;
use vars qw(@ISA $VERSION);
use Apache::Constants qw(:common);
use GD ();
use Image::Size qw(imgsize);
use LWP::UserAgent ();

@ISA = qw(LWP::UserAgent);
$VERSION = '1.00';

my $UA = __PACKAGE__->new;
$UA->agent(join "/", __PACKAGE__, $VERSION);

my $Ad = join "|", qw{ads? advertisement banner};

sub handler {
    my($r) = @_;
    return DECLINED unless $r->proxyreq;
    $r->handler("perl-script"); #ok, let's do it
    $r->push_handlers(PerlHandler => \&proxy_handler);
    return OK;
}

sub proxy_handler {
    my($r) = @_;

    my $request = HTTP::Request->new($r->method, $r->uri);

    $r->headers_in->do(sub {
        $request->header(@_);
    });

    # copy POST data, if any
    if($r->method eq 'POST') {
        my $len = $r->header_in('Content-length');

        my $buf;
        $r->read($buf, $len);
        $request->content($buf);
        $request->content_type($r->content_type);
    }
}
```

```

my $response = $UA->request($request);
$r->content_type($response->header('Content-type'));

#feed response back into our request_rec*
$r->status($response->code);
$r->status_line(join " ", $response->code, $response->message);
$response->scan(sub {
    $r->header_out(@_);
});

if ($r->header_only) {
    $r->send_http_header();
    return OK;
}

my $content = \"$response->content;
if($r->content_type =~ /^image/ and $r->uri =~ /\b($Ad)\b/i) {
    block_ad($content);
    $r->content_type("image/gif");
}

$r->content_type('text/html') unless $$content;
$r->send_http_header;
$r->print($$content || $response->error_as_HTML);

return OK;
}

sub block_ad {
    my $data = shift;
    my($x, $y) = imgsize($data);

    my $im = GD::Image->new($x,$y);

    my $white = $im->colorAllocate(255,255,255);
    my $black = $im->colorAllocate(0,0,0);
    my $red = $im->colorAllocate(255,0,0);

    $im->transparent($white);
    $im->string(GD::gdLargeFont(),5,5,"Blocked Ad",$red);
    $im->rectangle(0,0,$x-1,$y-1,$black);

    $$data = $im->gif;
}

1;

```

Another way of doing this module would be to scan all proxied HTML files for tags containing one of the verboten URLs, then replacing the src attribute with a transparent GIF of our own. However, unless the tag contained width and height attributes, we wouldn't be able to return a GIF of the correct size -- unless we were to go hunting for the GIF with LWP, in which case we might as well do it this way.

3.4.6 Customized Logging

After Apache handles a transaction, it passes all the information about the transaction to the log handler. The default log handler writes out lines to the log file. With `mod_perl`, you can install your own log handler to do customized logging.

3.4.6.1 Send E-Mail When a Particular Page Gets Hit

Script III.6.1 installs a log handler which watches over a page or set of pages. When someone fetches a watched page, the log handler sends off an e-mail to notify someone (probably the owner of the page) that the page has been read.

To activate the module, just attach a `PerlLogHandler` to the `<Location>` or `<Files>` you wish to watch. For example:

```
<Location /~lstein>
    PerlLogHandler Apache::LogMail
    PerlSetVar mailto lstein@cshl.org
</Location>
```

The "mailto" directive specifies the name of the recipient(s) to notify.

```
Script III.6.1: Apache::LogMail
-----
package Apache::LogMail;
use Apache::Constants ':common';

sub handler {
    my $r = shift;
    my $mailto = $r->dir_config('mailto');
    return DECLINED unless $mailto;
    my $request = $r->the_request;
    my $uri = $r->uri;
    my $agent = $r->header_in("User-agent");
    my $bytes = $r->bytes_sent;
    my $remote = $r->get_remote_host;
    my $status = $r->status_line;
    my $date = localtime;
    unless (open (MAIL,"|/usr/lib/sendmail -oi -t")) {
        $r->log_error("Couldn't open mail: $!");
        return DECLINED;
    }
    print MAIL <<END;
    To: $mailto
    From: Mod Perl <webmaster>
    Subject: Somebody looked at $uri

    At $date, a user at $remote looked at
    $uri using the $agent browser.

    The request was $request,
    which resulted returned a code of $status.

    $bytes bytes were transferred.
```



```

END
    close MAIL;
    return OK;
}
1;

```

3.4.6.2 Writing Log Information Into a Relational Database

Coming full circle, Script III.6.2 shows a module that writes log information into a DBI database. The idea is similar to Script I.1.9, but there's now no need to open a pipe to an external process. It's also a little more efficient, because the log data fields can be recovered directly from the Apache request object, rather than parsed out of a line of text. Another improvement is that we can set up the Apache configuration files so that only accesses to certain directories are logged in this way.

To activate, add something like this to your configuration file: `PerlLogHandler Apache::LogDBI`

Or, to restrict special logging to accesses of files in below the URL `"/lincoln_logs"` add this:

```

<Location /lincoln_logs>
    PerlLogHandler Apache::LogDBI
</Location>

Script III.6.2: Apache::LogDBI
-----
package Apache::LogDBI;
use Apache::Constants 'common';

use strict 'vars';
use vars qw($DB $STH);
use DBI;
use POSIX 'strftime';

use constant DSN      => 'dbi:mysql:www';
use constant DB_TABLE => 'access_log';
use constant DB_USER  => 'nobody';
use constant DB_PASSWD => '';

$DB = DBI->connect(DSN,DB_USER,DB_PASSWD) || die DBI->errstr;
$STH = $DB->prepare("INSERT INTO ${DB_TABLE} VALUES(?,?,?,?,?,?,?,?)")
    || die $DB->errstr;

sub handler {
    my $r = shift;
    my $date   = strftime('%Y-%m-%d %H:%M:%S',localtime);
    my $host   = $r->get_remote_host;
    my $method = $r->method;
    my $url    = $r->uri;
    my $user   = $r->connection->user;
    my $referer = $r->header_in('Referer');
    my $browser = $r->header_in("User-agent");
    my $status  = $r->status;
    my $bytes   = $r->bytes_sent;
    $STH->execute($date,$host,$method,$url,$user,
                 $browser,$referer,$status,$bytes);
}

```

```
        return OK;
    }

    1;
```

There are other alternatives which are more actively maintained available from the CPAN: `Apache::DBILogger` and `Apache::DBILogConfig`.

3.5 Conclusion

These tricks illustrate the true power of `mod_perl`; not only were Perl and Apache good friends from the start, thanks to Perl's excellent text-handling capacity, but when `mod_perl` is used, your complete access to the Apache API gives you unprecedented power in dynamic web serving.

To find more tips and tricks, look for modules on the CPAN, look through the `mod_perl` documentation, and also in the following books by Lincoln Stein:

- **"How to Set Up and Maintain a Web Site"**

General introduction to Web site care and feeding, with an emphasis on Apache. Addison-Wesley 1997.

Companion Web site at <http://www.genome.wi.mit.edu/WWW/>

- **"Web Security, a Step-by-Step Reference Guide"**

How to keep your Web site free from thieves, vandals, hooligans and other yahoos. Addison-Wesley 1998.

Companion Web site at <http://www.w3.org/Security/Faq/>

- **"The Official Guide to Programming with CGI.pm"**

Everything I know about CGI.pm (and some things I don't!). John Wiley & Sons, 1998.

Companion Web site at <http://www.wiley.com/legacy/compbooks/stein/>

- **"Writing Apache Modules in Perl and C"**

Co-authored with Doug MacEachern. O'Reilly & Associates.

Companion Web site at <http://www.modperl.com/>

- **WebTechniques Columns**

I write a monthly column for WebTechniques magazine (now New Architect). You can find back-issues and reprints at <http://www.webtechniques.com/>

- **The Perl Journal Columns**

I write a quarterly column for TPJ. Source code listings are available at <http://www.tpj.com/>

3.6 Maintainers

The maintainer is the person(s) you should contact with updates, corrections and patches.

- Per Einar Ellefsen <per.einar (at) skynet.be>

3.7 Authors

- Lincoln Stein <lstein (at) cshl.org>

Only the major authors are listed above. For contributors see the Changes file.

4 Workarounds for some known bugs in browsers.

4.1 Description

Unfortunately for web programmers, browser bugs are not uncommon, and sometimes we have to deal with them; refer to this chapter for some known bugs and how you can work around them.

4.2 Same Browser Requests Serialization

The following feature/bug mostly affects developers.

Certain browsers will serialize requests to the same URL if accessed from different windows. For example if you have a CGI script that does:

```
for (1..100) {  
    print "$$: $_\n";  
    warn  "$$: $_\n";  
    sleep 1;  
}
```

And two concurrent requests are issued from different windows of the same browser (for those browsers that have this bug/feature), the browser will actually issue only one request and won't run the second request till the first one is finished. The debug printing to the `error_log` file helps to understand the serialization issue.

Solution? Find a UA that doesn't have this feature, especially if a command line UA will do (LWP comes to mind). As of this writing, opera 6, mozilla 1.0 on linux have this problem, whereas konqueror 3 and lynx don't.

4.3 Preventing QUERY_STRING from getting corrupted because of &entity key names

In a URL which contains a query string, if the string has multiple parts separated by ampersands and it contains a key named "reg", for example

`http://example.com/foo.pl?foo=bar®=foobar`, then some browsers will interpret `®` as an SGML entity and encode it as `®`. This will result in a corrupted QUERY_STRING. If you encounter this problem, then either you should avoid using such keys or you should separate parameter pairs with `;` instead of `&`. CGI.pm, Apache::Request and `$r->args()` support a semicolon instead of an ampersand as a separator. So your URI should look like this: `http://example.com/foo.pl?foo=bar;reg=foobar`.

Note that this is only an issue when you are building your own URLs with query strings. It is not a problem when the URL is the result of submitting a form because the browsers *have* to get that right.

4.4 IE 4.x does not re-post data to a non-port-80 URL

One problem with publishing 8080 port numbers (or so I have been told) is that IE 4.x has a bug when re-posting data to a non-port-80 URL. It drops the port designator and uses port 80 anyway.

See Publishing Port Numbers other than 80.

4.5 Internet Explorer disregards your ErrorDocuments

Many users stumble upon a common problem related to MS Internet Explorer: if your error response, such as when using `ErrorDocument 500` or `$r->custom_response`, is too short (which might often be the case because you aren't very inspired when writing error messages), Internet Explorer completely disregards it and replaces it with its own standard error page, even though everything has been sent correctly by the server and received by the browser.

The solution to this is quite simple: your content needs to be at least 512 bytes. Microsoft describes some solutions to this *problem* here: <http://support.microsoft.com/support/kb/articles/Q294/8/07.ASP> . The easiest solution under Perl is to do something like this:

```
# write your HTML headers
print "<!-- ", "_" x 513, " -->";
# write out the rest of your HTML
```

Effectively, your content will be long enough, but the user won't notice any additional content. If you're doing this with static pages, just insert a long enough comment inside your file to make it large enough, which will have the same effect.

4.6 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

4.7 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

5 Web Content Compression FAQ

5.1 Basics of Content Compression

Compression of outgoing traffic from web servers is beneficial for clients who get quicker responses, as well as for providers who experience less consumption of bandwidth.

Recently content compression for web servers has been provided mainly through use of the `gzip` format. Other (non perl) modules are available that provide so-called `deflate` compression. Both approaches are currently very similar and use the LZ77 algorithm combined with Huffman coding. Luckily for us, there is no real need to understand all the details of the obscure underlying mathematics in order to compress outbound content. Apache handlers available from CPAN can usually do the dirty work for us. Content compression is addressed through the proper configuration of appropriate handlers in the `httpd.conf` file.

Compression by its nature is a content filter: It always takes its input as plain ASCII data that it converts to another binary form and outputs the result to some destination. That's why every content compression handler usually belongs to a particular chain of handlers within the content generation phase of the request-processing flow.

A chain of handlers is one more common term that is good to know about when you plan to compress data. There are two of them recently developed for Apache 1.3.X: `Apache::OutputChain` and `Apache::Filter`. We have to keep in mind that the compression handler developed for one chain usually fails inside another.

Another important point deals with the order of execution of handlers in a particular chain. It's pretty straightforward in `Apache::Filter`. For example, when you configure

```
PerlModule Apache::Filter
<Files ~ "\.blah">
    SetHandler perl-script
    PerlSetVar Filter On
    PerlHandler Filter1 Filter2 Filter3
</Files>
```

the content will go through `Filter1` first, then the result will be filtered by `Filter2`, and finally `Filter3` will be invoked to make the final changes in outgoing data.

However, when you configure

```
PerlModule Apache::OutputChain
PerlModule Apache::GzipChain
PerlModule Apache::SSIChain
PerlModule Apache::PassHtml
<Files *.html>
SetHandler perl-script
    PerlHandler Apache::OutputChain Apache::GzipChain Apache::SSIChain Apache::PassHtml
</Files>
```

execution begins with `Apache::PassHtml`. Then the content will be processed with `Apache::SSIChain` and finally with `Apache::GzipChain`. `Apache::OutputChain` will not be involved in content processing at all. It is there only for the purpose of joining other handlers within the chain.

It is important to remember that the content compression handler should always be the last executable handler in any chain.

Another important problem of practical implementation of web content compression deals with the fact that some buggy web clients declare the ability to receive and decompress gzipped data in their HTTP requests, but fail to keep their promises when an actual compressed response arrives. This problem is addressed through the implementation of the `Apache::CompressClientFixup` handler. This handler serves the `fixup` phase of the request-processing flow. It is compatible with all known compression handlers and is available from CPAN.

5.2 Q: Why it is important to compress web content?

5.2.1 A: *Reduced equipment costs and the competitive advantage of dramatically faster page loads.*

Web content compression noticeably increases delivery speed to clients and may allow providers to serve higher content volumes without increasing hardware expenditures. It visibly reduces actual content download time, a benefit most apparent to users of dialup and high-traffic connections.

5.3 Q: How much improvement can I expect?

5.3.1 A: *Effective compression can achieve increases in transmission efficiency from 3 to 20 times.*

The compression ratio is highly content-dependent. For example, if the compression algorithm is able to detect repeated patterns of characters, compression will be greater than if no such patterns exist. You can usually expect to realize an improvement between of 3 to 20 times on regular HTML, JavaScript, and other ASCII content. I have seen peak HTML file compression improvements in excess of more than 200 times, but such occurrences are infrequent. On the other hand I have never seen ratios of less than 2.5 times on text/HTML files. Image files normally employ their own compression techniques that reduce the advantage of further compression.

On May 21, 2002 Peter J. Cranstone wrote to the `mod_gzip@lists.over.net` mailing list:

"...With 98% of the world on a dial up modem, all they care about is how long it takes to download a page. It doesn't matter if it consumes a few more CPU cycles if the customer is happy. It's cheaper to buy a newer faster box, than it is to acquire new customers."

5.4 Q: How hard is it to implement content compression on an existing site?

5.5 Q: Does compression work with standard web browsers?

5.4.1 A: Implementing content compression on an existing site typically involves no more than installing and configuring an appropriate Apache handler on the web server.

This approach works in most of the cases I have seen. In some special cases you will need to take extra care with respect to the global architecture of your web application, but such cases may generally be readily addressed through various techniques. To date I have found no fundamental barriers to practical implementation of web content compression.

5.5 Q: Does compression work with standard web browsers?

5.5.1 A: Yes. No client side changes or settings are required.

All modern browser makers claim to be able to handle compressed content and are able to decompress it on the fly, transparent to the user. There are some known bugs in some old browsers, but these can be taken into account through appropriate configuration of the web server.

I strongly recommend use of the `Apache::CompressClientFixup` handler in your server configuration in order to prevent compression for known buggy clients.

5.6 Q: What software is required on the server side?

5.6.1 A: There are four known mod_perl modules/packages for the web content compression available to date for Apache 1.3.X (in alphabetical order):

- **Apache::Compress**

a mod_perl handler developed by Ken Williams (U.S.). `Apache::Compress` is capable to gzip output through `Apache::Filter`. This module accumulates all incoming data and then compresses the whole content body at once.

- **Apache::Dynagzip**

a mod_perl handler, developed by Slava Bizyayev -- a Russian programmer residing in the U.S. `Apache::Dynagzip` uses the gzip format to compress output through the `Apache::Filter` or through the internal Unix pipe.

`Apache::Dynagzip` is most useful when one needs to compress dynamic outbound web content (generated on the fly from databases, XML, etc.) when content length is not known at the time of the request.

`Apache::Dynagzip`'s features include:

- **Support for both HTTP/1.0 and HTTP/1.1.**
- **Control over the chunk size on HTTP/1.1 for on-the-fly content compression.**
- **Support for Perl, Java, or C/C++ CGI applications.**
- **Advanced control over the proxy cache with the configurable `Vary` HTTP header.**
- **Optional control over content lifetime in the client's local cache with the configurable `Expires` HTTP header.**
- **Optional support for server-side caching of the dynamically generated (and compressed) content.**
- **Optional extra-light compression**

removal of leading blank spaces and/or blank lines, which works for all browsers, including older ones that cannot uncompress gzip format.

- **`Apache::Gzip`**

an example of `mod_perl` filter developed by Lincoln Stein and Doug MacEachern for their book *Writing Apache Modules with Perl and C* (U.S.), which like `Apache::Compress` works through `Apache::Filter`. `Apache::Gzip` is not available from CPAN. The source code may be found on the book's companion web site at <http://www.modperl.com/>

- **`Apache::GzipChain`**

a `mod_perl` handler developed by Andreas Koenig (Germany), which compresses output through `Apache::OutputChain` using the gzip format.

`Apache::GzipChain` currently provides in-memory compression only. Using this module under perl-5.8 or higher is appropriate for Unicode data. UTF-8 data passed to `Compress::Zlib::memGzip()` are converted to raw UTF-8 before compression takes place. Other data are simply passed through.

5.7 Q: Is it possible to compress the output from `Apache::Registry` with `Apache::Dynagzip`?

5.7.1 A: *Yes, it is supposed to be pretty easy:*

If your page/application is initially configured like

```
<Directory /path/to/subdirectory>
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSendHeader On
  Options +ExecCGI
</Directory>
```

5.8 Q: Is it possible to compress the output from Mason-driven application with Apache::Dynagzip?

you might want just to replace it with the following:

```
PerlModule Apache::Filter
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
<Directory /path/to/subdirectory>
    SetHandler perl-script
    PerlHandler Apache::RegistryFilter Apache::Dynagzip
    PerlSendHeader On
    Options +ExecCGI
    PerlSetVar Filter On
    PerlFixupHandler Apache::CompressClientFixup
    PerlSetVar LightCompression On
</Directory>
```

You should be all set usually after that.

In more common cases you need to replace the line

```
PerlHandler Apache::Registry
```

in your initial configuration file with the set of the following lines:

```
PerlHandler Apache::RegistryFilter Apache::Dynagzip
PerlSetVar Filter On
PerlFixupHandler Apache::CompressClientFixup
```

You might want to add optionally

```
PerlSetVar LightCompression On
```

to reduce the size of the stream even for clients incapable to speak gzip (like *Microsoft Internet Explorer* over HTTP/1.0).

Finally, make sure you have somewhere declared

```
PerlModule Apache::Filter
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
```

This basic configuration uses many defaults. See `Apache::Dynagzip` POD for further thin tuning if required.

5.8 Q: Is it possible to compress the output from Mason-driven application with Apache::Dynagzip?

5.8.1 A: Yes. HTML::Mason::ApacheHandler is compatible with Apache::Filter chain.

If your application is initially configured like

```
PerlModule HTML::Mason::ApacheHandler
<Directory /path/to/subdirectory>
  <FilesMatch "\.html$">
    SetHandler perl-script
    PerlHandler HTML::Mason::ApacheHandler
  </FilesMatch>
</Directory>
```

you might want just to replace it with the following:

```
PerlModule HTML::Mason::ApacheHandler
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
<Directory /path/to/subdirectory>
  <FilesMatch "\.html$">
    SetHandler perl-script
    PerlHandler HTML::Mason::ApacheHandler Apache::Dynagzip
    PerlSetVar Filter On
    PerlFixupHandler Apache::CompressClientFixup
    PerlSetVar LightCompression On
  </FilesMatch>
</Directory>
```

You should be all set safely after that.

In more common cases you need to replace the line

```
PerlHandler HTML::Mason::ApacheHandler
```

in your initial configuration file with the set of the following lines:

```
PerlHandler HTML::Mason::ApacheHandler Apache::Dynagzip
PerlSetVar Filter On
PerlFixupHandler Apache::CompressClientFixup
```

You might want to add optionally

```
PerlSetVar LightCompression On
```

to reduce the size of the stream even for clients incapable to speak gzip (like *Microsoft Internet Explorer* over HTTP/1.0).

Finally, make sure you have somewhere declared

```
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
```

5.9 Q: Why is it important to keep control over chunk size?

This basic configuration uses many defaults. See `Apache::Dynagzip` POD for further thin tuning.

5.9 Q: Why is it important to keep control over chunk size?

5.9.1 A: *It helps to reduce the latency of the response.*

`Apache::Dynagzip` is the only handler to date that begins transmission of compressed data as soon as the initial uncompressed pieces of data arrive from their source, at a time when the source process may not even have completed generating the full document it is sending. Transmission can therefore be taking place concurrent with creation of later document content.

This feature is mainly beneficial for HTTP/1.1 requests, because HTTP/1.0 does not support chunks.

I would also mention that the internal buffer in `Apache::Dynagzip` always prevents Apache from the creating too short chunks over HTTP/1.1, or from transmitting too short pieces of data over HTTP/1.0.

5.10 Q: Are there any content compression solutions for vanilla Apache 1.3.X?

5.10.1 A: *Yes, There are two compression modules written in C that are available for vanilla Apache 1.3.X:*

- **mod_deflate**

an Apache handler written in C by Igor Sysoev (Russia).

- **mod_gzip**

an Apache handler written in C. Original author: Kevin Kiley, *Remote Communications, Inc.* (U.S.)

Both of these modules support HTTP/1.0 only.

5.11 Q: Can I compress the output of my site at the application level?

5.11.1 A: *Yes, if your web server is CGI/1.1 compatible and allows you to create specific HTTP headers from your application, or when you use an application framework that carries its own handler capable of compressing outbound data.*

For example, vanilla Apache 1.3.X is CGI/1.1 compatible. It allows development of CGI scripts/programs that might be generating compressed outgoing streams accomplished with specific HTTP headers.

Alternatively, on mod_perl enabled Apache some application environments carry their own compression code that could be activated through the appropriate configurations:

Apache::ASP does this with the CompressGzip setting;

Apache::AxKit uses the AxGzipOutput setting to do this.

See particular package documentation for details.

5.12 Q: Are there any content compression solutions for Apache-2?

5.12.1 A: Yes, a core compression module written in C, *mod_deflate*, has recently become available for Apache-2.

mod_deflate for Apache-2 is written by Ian Holsman (USA).

This module supports HTTP/1.1 and is filters compatible.

Despite its name mod_deflate for Apache-2 provides gzip-encoded content. It contains a set of configuration options sufficient to keep control over all recently known buggy web clients.

5.13 Q: When Apache::Dynagzip is supposed to be ported to Apache-2?

5.13.1 A: There no recent plans to port Apache::Dynagzip to Apache-2:

mod_deflate for Apache-2 seems to be capable to provide all basic functionality required for dynamic content compression:

- **This module supports flushing over HTTP/1.1**
- **It is filters compatible.**
- **It has a set of configuration options to keep control over the buggy clients.**

The rest of the main Apache::Dynagzip options could be easily addressed through the implementation of pretty tiny and specific accomplishing filters.

5.14 Q: Where can I read the original descriptions of `gzip` and `deflate` formats?

5.14 Q: Where can I read the original descriptions of `gzip` and `deflate` formats?

5.14.1 A: *`gzip` format is published as `rfc1952`, and `deflate` format is published as `rfc1951`.*

You can find many mirrors of RFC archives on the Internet. Try, for instance, my favorite at <http://www.ietf.org/rfc.html>

5.15 Q: Are there any known compression problems with specific browsers?

5.15.1 A: *Yes, Netscape 4 has problems with compressed cascading style sheets and JavaScript files.*

You can use `Apache::CompressClientFixup` to disable compression for these files dynamically. `Apache::Dynagzip` is capable of providing so-called `light` compression for these files.

5.16 Q: Where can I find more information about the compression features of modern browsers?

5.16.1 A: *Michael Schroepf maintains a highly valuable site*

Try it at http://www.schroepf.net/projekte/mod_gzip/browser.htm

5.17 Acknowledgments

I highly appreciate efforts of Dan Hansen done in order to make this text better English...

5.18 Maintainers

The maintainer is the person you should contact with updates, corrections and patches.

- Slava Bizyayev <slava (at) cpan.org>

5.19 Authors

- Slava Bizyayev <slava (at) cpan.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

Tutorials	1
Building a Large-Scale E-commerce site with Apache and mod_perl	3
1 Building a Large-Scale E-commerce site with Apache and mod_perl	3
1.1 Description	4
1.2 Common Myths	4
1.3 Perl Saves	4
1.3.1 Roll Your Own Application Server	4
1.4 Case Study: eToys.com	5
1.4.1 Apache::PerlRun to the Rescue	5
1.4.2 Planning the New Architecture	6
1.5 Surviving Christmas 2000	6
1.5.1 The Architecture	6
1.5.2 Proxy Servers	7
1.5.3 Application Servers	8
1.5.4 Search servers	9
1.6 Load Balancing and Failover	10
1.7 Code Structure	11
1.8 Caching	11
1.9 Session Tracking	12
1.10 Security	12
1.11 Exception Handling	13
1.12 Templates	14
1.13 Controller Example	14
1.14 Performance Tuning	16
1.15 Trap: Nested Exceptions	17
1.16 Berkeley DB	17
1.17 Valuable Tools	18
1.18 Do Try This at Home	19
1.19 An Open Source Success Story	19
1.20 Maintainers	19
1.21 Authors	20
Choosing a Templating System	21
2 Choosing a Templating System	21
2.1 Description	22
2.2 Introduction	22
2.2.1 On A Personal Note	22
2.3 Why Use Templates?	22
2.3.1 Consistency of Appearance	22
2.3.2 Reusability	23
2.3.3 Better Isolation from Changes	23
2.3.4 Division of Labor	23
2.4 What Are the Differences?	23
2.4.1 Execution Models	23
2.4.2 Languages	24

Table of Contents:

2.4.3	Parsers and Caching	26
2.4.4	Application Frameworks vs. Just Templates	27
2.4.4.1	URL Mapping	27
2.4.4.2	Session Tracking	27
2.4.4.3	Output Caching	27
2.4.4.4	Form Handling	27
2.4.4.5	Debugging	27
2.5	The Contenders	28
2.5.1	SSI	28
2.5.2	HTML::Mason	28
2.5.3	HTML::Embperl	29
2.5.4	Apache::AxKit	30
2.5.5	Apache::ASP	31
2.5.6	Text::Template	32
2.5.7	Template Toolkit	33
2.5.8	HTML::Template	34
2.5.9	HTML_Tree	35
2.5.10	Apache::XPP	35
2.5.11	ePerl	35
2.5.12	CGI::FastTemplate	35
2.6	Performance	35
2.6.1	CGI Performance Concerns	36
2.7	Matrix	36
2.8	Updates	37
2.9	Maintainers	37
2.10	Authors	37
	Cute Tricks With Perl and Apache	38
3	Cute Tricks With Perl and Apache	38
3.1	Description	39
3.2	Part I: Tricks with CGI.pm	39
3.2.1	Dynamic Documents	39
3.2.1.1	Making HTML look beautiful	39
3.2.1.2	Making HTML concise	40
3.2.1.3	Making Interactive Forms	41
3.2.2	Making Stateful Forms	41
3.2.2.1	Keeping State with Cookies	43
3.2.3	Creating Non-HTML Types	45
3.2.4	Document Translation	48
3.2.4.1	Smart Redirection	51
3.2.5	File Uploads	52
3.3	Part II: Web Site Care and Feeding	53
3.3.1	Logs! Logs! Logs!	53
3.3.1.1	Log rotation	53
3.3.1.2	Log rotation and archiving	54
3.3.1.3	Log rotation, compression and archiving	54
3.3.1.4	Log Parsing	55
3.3.1.5	Offline Reverse DNS Resolution	57

3.3.1.6 Detecting Robots	58
3.3.1.7 Logging to syslog	59
3.3.1.8 Logging to a relational database	60
3.3.2 My server fell down and it can't get up!	61
3.3.2.1 Monitoring a local server	61
3.3.2.2 Monitoring a remote server	62
3.3.2.3 Resurrecting Dead Servers	63
3.3.3 Site Replication and Mirroring	64
3.3.3.1 Mirroring Single Pages	64
3.3.3.2 Mirroring a Document Tree	65
3.3.3.3 Checking for Bad Links	66
3.3.4 Load balancing	68
3.3.5 Torture Testing a Server	70
3.4 Part III: mod_perl -- Faster Than a Speeding Bullet	72
3.4.1 Creating Dynamic Pages	72
3.4.2 File Filters	73
3.4.2.1 Adding a Canned Footer to Every Page	73
3.4.2.2 Dynamic Navigation Bar	74
3.4.2.3 On-the-Fly Compression	77
3.4.3 Access Control	80
3.4.4 Authentication and Authorization	81
3.4.4.1 Authentication with NIS	82
3.4.4.2 Anonymous Authentication	83
3.4.4.3 Gender-Based Authorization	84
3.4.5 Proxy Services	85
3.4.5.1 A Banner Ad Blocker	85
3.4.6 Customized Logging	88
3.4.6.1 Send E-Mail When a Particular Page Gets Hit	88
3.4.6.2 Writing Log Information Into a Relational Database	89
3.5 Conclusion	90
3.6 Maintainers	91
3.7 Authors	91
Workarounds for some known bugs in browsers.	92
4 Workarounds for some known bugs in browsers.	92
4.1 Description	93
4.2 Same Browser Requests Serialization	93
4.3 Preventing QUERY_STRING from getting corrupted because of &entity key names	93
4.4 IE 4.x does not re-post data to a non-port-80 URL	94
4.5 Internet Explorer disregards your ErrorDocuments	94
4.6 Maintainers	94
4.7 Authors	94
Web Content Compression FAQ	95
5 Web Content Compression FAQ	95
5.1 Basics of Content Compression	96
5.2 Q: Why it is important to compress web content?	97
5.2.1 A: Reduced equipment costs and the competitive advantage of dramatically faster page loads.	97

5.3 Q: How much improvement can I expect?	97
5.3.1 A: Effective compression can achieve increases in transmission efficiency from 3 to 20 times.	97
5.4 Q: How hard is it to implement content compression on an existing site?	97
5.4.1 A: Implementing content compression on an existing site typically involves no more than installing and configuring an appropriate Apache handler on the web server.	98
5.5 Q: Does compression work with standard web browsers?	98
5.5.1 A: Yes. No client side changes or settings are required.	98
5.6 Q: What software is required on the server side?	98
5.6.1 A: There are four known mod_perl modules/packages for the web content compression available to date for Apache 1.3.X (in alphabetical order):	98
5.7 Q: Is it possible to compress the output from <code>Apache::Registry</code> with <code>Apache::Dynagzip</code> ?	99
5.7.1 A: Yes, it is supposed to be pretty easy:	99
5.8 Q: Is it possible to compress the output from Mason-driven application with <code>Apache::Dynagzip</code> ?	100
5.8.1 A: Yes. <code>HTML::Mason::ApacheHandler</code> is compatible with <code>Apache::Filter</code> chain.	101
5.9 Q: Why is it important to keep control over chunk size?	102
5.9.1 A: It helps to reduce the latency of the response.	102
5.10 Q: Are there any content compression solutions for vanilla Apache 1.3.X?	102
5.10.1 A: Yes, There are two compression modules written in C that are available for vanilla Apache 1.3.X:	102
5.11 Q: Can I compress the output of my site at the application level?	102
5.11.1 A: Yes, if your web server is CGI/1.1 compatible and allows you to create specific HTTP headers from your application, or when you use an application framework that carries its own handler capable of compressing outbound data.	102
5.12 Q: Are there any content compression solutions for Apache-2?	103
5.12.1 A: Yes, a core compression module written in C, <code>mod_deflate</code> , has recently become available for Apache-2.	103
5.13 Q: When <code>Apache::Dynagzip</code> is supposed to be ported to Apache-2?	103
5.13.1 A: There no recent plans to port <code>Apache::Dynagzip</code> to Apache-2:	103
5.14 Q: Where can I read the original descriptions of <code>gzip</code> and <code>deflate</code> formats?	104
5.14.1 A: <code>gzip</code> format is published as <code>rfc1952</code> , and <code>deflate</code> format is published as <code>rfc1951</code>	104
5.15 Q: Are there any known compression problems with specific browsers?	104
5.15.1 A: Yes, Netscape 4 has problems with compressed cascading style sheets and JavaScript files.	104
5.16 Q: Where can I find more information about the compression features of modern browsers?	104
5.16.1 A: Michael Schroepf maintains a highly valuable site	104
5.17 Acknowledgments	104
5.18 Maintainers	104
5.19 Authors	105