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.com, 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

Machine 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

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

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

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 send 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

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-E<gt>new(NAME =E<gt> $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 it's 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 it's 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 appeared. We didn't use one of these, but they have a similar design to what we did and may prove useful to you if you want to take an MVC approach with your system.

Some of the most interesting tools for MVC web development in Perl include `Apache::PageKit`, `OpenInteract2`, `CGI::Application`, `Maypole`, and `Catalyst`. There isn't room here to get deeply into the differences between these tools, but watch for an article comparing these frameworks in the future.

If you want a ready-to-use cache module, there are several on CPAN now. The most popular is the `Cache::Cache` framework, which can use files or shared memory for storage. Since the original writing of this article, newer and faster options have appeared, particularly `Cache::FastMmap` and `Cache::Memcached`.

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.

Table of Contents: