APPENDIX B

# Apache Perl Modules

Many third-party modules have been written to extend mod_perl's core functionality. They may be distributed with the mod_perl source code, or they may be available from CPAN. In this chapter we will attempt to group these modules based on their functionality. Some modules will be discussed in depth, but others will be touched on only briefly.

Since most of these modules are continually evolving, the moment this book is published much of the information in it will be out of date. For this reason, you should refer to the modules' manpages when you start using them; that's where you will find the most up-to-date documentation.

We will consider modules in the following groups:

*Development*
: Modules used mainly during the development process

*Debugging*
: Modules that assist in code debugging

*Control and monitoring*
: Modules to help you monitor the production server and take care of any problems as soon as they appear

*Server configuration*
: Modules used in server configuration

*Authentication*
: Modules used to facilitate authentication

*Authorization*
: Modules used to facilitate authorization

*Access*
: Modules used during the access-verification phase

*Type handlers*
: Modules used as `PerlTypeHandlers`

*Trans handlers*
: Modules used as `PerlTransHandlers`

*Fixup Handlers*
: Modules used as `PerlFixupHandlers`

*Generic content-generation phase*
: Generic modules that assist during the content-generation phase

*Application-specific content generation phase*
: Non–general-purpose content generators

*Database*
: Database-specific modules

*Toolkits and framework for content generation and other phases*
: Mostly large toolkits and frameworks built on top of mod_perl

*Output filters and layering*
: Modules that filter output from the content generation stage

*Logging-phase handlers*
: Modules that assist during the logging stage

*Core Apache*
: Modules that interface with core mod_perl

*Miscellaneous*
: Modules that don't fit into any of the above categories

# Development-Stage Modules

The following modules are mainly useful during the code-development cycle. Some of them can also be useful in the production environment.

## Apache::Reload—Automatically Reload Changed Modules

`Apache::Reload` is used to make specific modules reload themselves when they have changed. It's also very useful for mod_perl module development.

Covered in Chapter 6.

Available from CPAN. See the module manpage for more information.

## Apache::PerlVINC—Allow Module Versioning in <Location> and <VirtualHost> blocks

This module makes it possible to have different `@INC` values for different `<VirtualHost>`s, `<Location>`s, and equivalent configuration blocks.

Suppose two versions of Apache::Status are being hacked on the same server. In this configuration:

```
PerlModule Apache::PerlVINC

<Location /status-dev/perl>
    SetHandler       perl-script
    PerlHandler      Apache::Status

    PerlINC          /home/httpd/dev/lib
    PerlFixupHandler Apache::PerlVINC
    PerlVersion      Apache/Status.pm
</Location>

<Location /status/perl>
    SetHandler       perl-script
    PerlHandler      Apache::Status

    PerlINC          /home/httpd/prod/lib
    PerlFixupHandler Apache::PerlVINC
    PerlVersion      Apache/Status.pm
</Location>
```

Apache::PerlVINC is loaded and then two different locations are specified for the same handler Apache::Status, whose development version resides in */home/httpd/dev/lib* and production version in */home/httpd/prod/lib*.

If a request for */status/perl* is issued (the latter configuration section), the fixup handler will internally do:

```
delete $INC{"Apache/Status.pm"};
unshift @INC, "/home/httpd/prod/lib";
require Apache::Status;
```

which will load the production version of the module, which will in turn be used to process the request.

If on the other hand the request is for */status-dev/perl* (the former configuration section), a different path (*/home/httpd/dev/lib*) will be prepended to @INC:

```
delete $INC{"Apache/Status.pm"};
unshift @INC, "/home/httpd/dev/lib";
require Apache::Status;
```

It's important to be aware that a changed @INC is effective only inside the <Location> block or a similar configuration directive. Apache::PerlVINC subclasses the PerlRequire directive, marking the file to be reloaded by the fixup handler, using the value of PerlINC for @INC. That's local to the fixup handler, so you won't actually see @INC changed in your script.

Additionally, modules with different versions can be unloaded at the end of the request, using the PerlCleanupHandler:

```
<Location /status/perl>
    SetHandler          perl-script
    PerlHandler         Apache::Status

    PerlINC             /home/httpd/prod/lib
    PerlFixupHandler    Apache::PerlVINC
    PerlCleanupHandler Apache::PerlVINC
    PerlVersion         Apache/Status.pm
</Location>
```

Also note that `PerlVersion` affects things differently depending on where it is placed. If it is placed inside a `<Location>` or a similar block section, the files will be reloaded only on requests to that location. If it is placed in a server section, all requests to the server or virtual hosts will have these files reloaded.

As you can guess, this module slows down the response time because it reloads some modules on a per-request basis. Hence, this module should be used only in a development environment, not in production.

If you need to do the same in production, a few techniques are suggested in Chapter 4.

Available from CPAN. See the module manpage for more information.

## Apache::DProf—Hook Devel::DProf into mod_perl

Covered in Chapter 9.

Available from CPAN. See the module manpage for more information.

## Apache::SmallProf—Hook Devel::SmallProf into mod_perl

Covered in Chapter 9.

Available from CPAN. See the module manpage for more information.

## Apache::FakeRequest—Fake Request Object for Debugging

Covered in Chapter 21.

Available from CPAN. See the module manpage for more information.

## Apache::test—Facilitate Testing of Apache::* Modules

This module helps authors of `Apache::*` modules write test suites that can query a running Apache server with mod_perl and their modules loaded into it. Its functionality is generally separated into: (a) methods that go in a *Makefile.PL* file to configure, start, and stop the server; and (b) methods that go into one of the test scripts to make HTTP queries and manage the results.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Modules to Aid Debugging

The following modules are used mainly when something is not working properly and needs to be debugged. Unless your bug is very hard to reproduce and the production environment is required to reproduce the conditions that will trigger the bug, these modules should not be used in production.

### Apache::DB—Hooks for the Interactive Perl Debugger

Allows developers to interactively debug mod_perl.

Covered in Chapter 9.

Available from CPAN. See the module manpage for more information.

### Apache::Debug—Utilities for Debugging Embedded Perl Code

Covered in Chapter 21.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::DebugInfo—Send Debug Information to Client

Available from CPAN. See the module manpage for more information.

### Apache::Leak—Module for Tracking Memory Leaks in mod_perl Code

Covered in Chapter 14.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::Peek—A Data Debugging Tool for the XS Programmer

Covered in Chapter 10.

Available from CPAN. See the module manpage for more information.

### Apache::Symbol—Avoid the Mandatory 'Subroutine Redefined' Warning

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::Symdump—Symbol Table Snapshots

Covered in Chapter 21.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Control and Monitoring Modules

### Apache::Watchdog::RunAway—Hanging Processes Monitor and Terminator

Covered in Chapter 5.

Available from CPAN. See the module manpage for more information.

### Apache::VMonitor—Visual System and Apache Server Monitor

Covered in Chapter 5.

Available from CPAN. See the module manpage for more information.

### Apache::SizeLimit—Limit Apache httpd Processes

This module allows you to kill off Apache processes if they grow too large or if they share too little of their memory. It's similar to `Apache::GTopLimit`.

Covered in Chapter 14.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::GTopLimit—Limit Apache httpd Processes

This module allows you to kill off Apache processes if they grow too large or if they share too little of their memory. It's similar to `Apache::SizeLimit`.

Covered in Chapter 14.

Available from CPAN. See the module manpage for more information.

### Apache::TimedRedirect—Redirect URLs for a Given Time Period

`Apache::TimedRedirect` is a mod_perl `TransHandler` module that allows the configuration of a timed redirect. In other words, if a user enters a web site and the URI matches a regex *and* it is within a certain time period she will be redirected somewhere else.

This was first created to *politely* redirect visitors away from database-driven sections of a web site while the databases were being refreshed.

Available from CPAN. See the module manpage for more information.

### Apache::Resource—Limit Resources Used by httpd Children

`Apache::Resource` uses the `BSD::Resource` module, which uses the C function `setrlimit( )` to set limits on system resources such as memory and CPU usage.

Covered in Chapter 5.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::Status—Embedded Interpreter Status Information

The `Apache::Status` module provides various information about the status of the Perl interpreter embedded in the server.

Covered in Chapter 21.

Available from CPAN. See the module manpage for more information.

## Server Configuration Modules

### Apache::ModuleConfig—Interface to Configuration API

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::PerlSections—Utilities for Working with <Perl> Sections

`Apache::PerlSections` configures Apache entirely in Perl.

Covered in Chapter 4.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::httpd_conf—Generate an httpd.conf File

The `Apache::httpd_conf` module will generate a tiny *httpd.conf* file, which pulls itself back in via a `<Perl>` section. Any additional arguments passed to the `write()` method will be added to the generated *httpd.conf* file and will override those defaults set in the `<Perl>` section. This module is handy mostly for starting *httpd* servers to test mod_perl scripts and modules.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::src—Methods for Locating and Parsing Bits of Apache Source Code

This module provides methods for locating and parsing bits of Apache source code. For example:

```
my $src = Apache::src->new;
my $v = $src->httpd_version;
```

returns the server version. And:

```
my $dir = $src->dir;
-d $dir or die "can't stat $dir $!\n";
```

returns the top level directory where source files are located and then tests whether it can read it.

The `main()` method will return the location of *httpd.h*:

```
-e join "/", $src->main, "httpd.h" or die "can't stat httpd.h\n";
```

Other methods are available from this module.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::ConfigFile—Parse an Apache-Style httpd.conf Configuration File

This module parses *httpd.conf*, or any compatible configuration file, and provides methods for accessing the values from the parsed file.

Available from CPAN. See the module manpage for more information.

## Authentication-Phase Modules

The following modules make it easier to handle the authentication phase:

```
AuthenCache     Cache authentication credentials
AuthCookie      Authentication and authorization via cookies
AuthDigest      Authentication and authorization via digest scheme
AuthenDBI       Authenticate via Perl's DBI
AuthenIMAP      Authentication via an IMAP server
AuthenPasswdSrv External authentication server
AuthenPasswd    Authenticate against /etc/passwd
AuthLDAP        LDAP authentication module
AuthPerLDAP     LDAP authentication module (PerLDAP)
AuthenNIS       NIS authentication
AuthNISPlus     NIS Plus authentication/authorization
AuthenSmb       Authenticate against an NT server
AuthenURL       Authenticate via another URL
DBILogin        Authenticate to backend database
PHLogin         Authenticate via a PH database
```

All available from CPAN. See the module manpages for more information.

## Authorization-Phase Modules

The following modules make it easier to handle the authorization phase:

```
AuthCookie      Authentication and authorization via cookies
AuthzDBI        Group authorization via Perl's DBI
AuthzNIS        NIS authorization
AuthzPasswd     Authorize against /etc/passwd
```

All available from CPAN. See the module manpages for more information.

## Access-Phase Modules

The following modules are used during the access request phase:

```
AccessLimitNum  Limit user access by the number of requests
RobotLimit      Limit the access of robots
```

Available from CPAN. See the module manpages for more information.

### Stonehenge::Throttle—Limit Bandwith Consumption by IP Address

*http://www.stonehenge.com/merlyn/LinuxMag/col17.html*

The source code to `Stonehenge::Throttle` is available from *http://www.stonehenge.com/merlyn/LinuxMag/col17.listing.txt*.

## Type Handlers

### Apache::MimeXML—mod_perl Mime Encoding Sniffer for XML Files

This module is an XML `content-type` sniffer. It reads the encoding attribute in the XML declaration and returns an appropriate content-type heading. If no encoding declaration is found it returns *utf-8* or *utf-16*, depending on the specific encoding.

Available from CPAN. See the module manpage for more information.

### Apache::MIMEMapper—Associates File Extensions with PerlHandlers

`Apache::MIMEMapper` extends the core `AddHandler` directive to allow you to dispatch different `PerlHandlers` based on the file extension of the requested resource.

Available from CPAN. See the module manpage for more information.

## Trans Handlers

### Apache::AddHostPath—Adds Some or All of the Hostname and Port to the URI

This module transforms the requested URI based on the hostname and port number from the HTTP request header. It allows you to manage an arbitrary number of domains and subdomains all pointing to the same document root but for which you want a combination of shared and distinct files.

Essentially the module implements Apache's URI-translation phase by attempting to use some or all of the URL hostname and port number as the base of the URI. It simply does file and directory existence tests on a series of URIs (from most-specific to least-specific) and sets the URI to the most specific match.

For example, if the request is:

```
URL: http://www.example.org:8080/index.html
URI: /index.html
```

`Apache::AddHostPath` would go through the following list of possible paths and set the new URI based on the first match that passes a `-f` or `-d` existence test:

```
$docRoot/org/example/www/8080/index.html
$docRoot/org/example/www/index.html
$docRoot/org/example/index.html
$docRoot/org/index.html
$docRoot/index.html
```

Available from CPAN. See the module manpage for more information.

## Apache::ProxyPass—implement ProxyPass in Perl

This module implements the Apache mod_proxy module in Perl. Based on `Apache::ProxyPassThru`.

Available from CPAN. See the module manpage for more information.

## Apache::ProxyPassThru—Skeleton for Vanilla Proxy

This module uses *libwww-perl* as its web client, feeding the response back into the Apache API `request_rec` structure. `PerlHandler` will be invoked only if the request is a proxy request; otherwise, your normal server configuration will handle the request.

If used with the `Apache::DumpHeaders` module it lets you view the headers from another site you are accessing.

Available from CPAN. See the module manpage for more information.

## Apache::Throttle—Speed-Based Content Negotiation

`Apache::Throttle` is a package designed to allow Apache web servers to negotiate content based on the speed of the connection. Its primary purpose is to transparently send smaller (lower resolution/quality) images to users with slow Internet connections, but it can also be used for many other purposes.

Available from CPAN. See the module manpage for more information.

## Apache::TransLDAP—Trans Handler Example

This module is an example of how you can create a trans handler. This particular example translates from a user's virtual directory on the server to the `labeledURI` attribute for the given user.

Available from CPAN. See the module manpage for more information.

# Fixup Handlers

## Apache::RefererBlock—Block Request Based Upon "Referer" Header

`Apache::RefererBlock` will examine each request. If the MIME type of the requested file is one of those listed in `RefBlockMimeTypes`, it will check the request's `Referer` header. If the referrer starts with one of the strings listed in `RefBlockAllowed`, access is granted. Otherwise, if there's a `RefBlockRedirect` directive for the referrer, a redirect is issued. If not, a "Forbidden" (403) error is returned.

Available from CPAN. See the module manpage for more information.

## Apache::Usertrack—Emulate the mod_usertrack Apache Module

As of this writing no documentation is available.

Available from CPAN.

# Generic Content-Generation Modules

These modules extend mod_perl functionality during the content-generation phase. Some of them can also be used during earlier phases.

## Apache::Registry and Apache::PerlRun

These two modules allow mod_cgi Perl scripts to run unaltered under mod_perl. They are covered throughout the book, mainly in Chapters 6 and 13.

See also the related `Apache::RegistryNG` and `Apache::RegistryBB` modules.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::RegistryNG—Apache::Registry New Generation

`Apache::RegistryNG` is almost the same as `Apache::Registry`, except that it uses filenames instead of URIs for namespaces. It also uses an object-oriented interface.

```
PerlModule Apache::RegistryNG
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::RegistryNG->handler
</Location>
```

The usage is just the same as `Apache::Registry`.

Apache::RegistryNG inherits from Apache::PerlRun, but the handler( ) is overriden. Apart from the handler( ), the rest of Apache::PerlRun contains all the functionality of Apache::Registry, broken down into several subclassable methods. These methods are used by Apache::RegistryNG to implement the exact same functionality as Apache::Registry, using the Apache::PerlRun methods.

There is no compelling reason to use Apache::RegistryNG over Apache::Registry, unless you want to add to or change the functionality of the existing Registry.pm. For example, Apache::RegistryBB is another subclass that skips the stat( ) call, Option +ExecCGI, and other checks performed by Apache::Registry on each request.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::RegistryBB—Apache::Registry Bare Bones

This works just like Apache::Registry, but it does not test the *x* bit (-x file test for executable mode), compiles the file only once (no stat( ) call is made for each request), skips the OPT_EXECCGI checks, and does not chdir( ) into the script's parent directory. It uses the object-oriented interface.

Configuration:

```
PerlModule Apache::RegistryBB
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::RegistryBB->handler
</Location>
```

The usage is just the same as Apache::Registry.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::Request (libapreq)—Generic Apache Request Library

This package contains modules for manipulating client request data via the Apache API with Perl and C. Functionality includes:

- Parsing *application/x-www-form-urlencoded* data
- Parsing *multipart/form* data
- Parsing HTTP cookies

The Perl modules are simply a thin XS layer on top of *libapreq*, making them a lighter and faster alternative to CGI.pm and CGI::Cookie. See the Apache::Request and Apache::Cookie documentation for more details and *eg/perl/* for examples.

`Apache::Request` and *libapreq* are tied tightly to the Apache API, to which there is no access in a process running under mod_cgi.

This module is mentioned in Chapters 6 and 13.

Available from CPAN. See the module manpage for more information.

### Apache::Dispatch—Call PerlHandlers with the Ease of Registry Scripts

`Apache::Dispatch` translates `$r->uri` into a class and method and runs it as a `PerlHandler`. Basically, this allows you to call `PerlHandlers` as you would Registry scripts, without having to load your *httpd.conf* file with a lot of `<Location >` tags.

Available from CPAN. See the module manpage for more information.

## Application-Specific Content-Generation Modules

### Apache::AutoIndex—Perl Replacement for the mod_autoindex and mod_dir Apache Modules

This module can completely replace the mod_dir and mod_autoindex standard directory-handling modules shipped with Apache.

Available from CPAN. See the module manpage for more information.

### Apache::WAP::AutoIndex—WAP Demonstration Module

This is a simple module to demonstrate the use of `CGI::WML` to create a WML (wireless) file browser using mod_perl. It was written to accompany an article in the *Perl Journal* (Issue 20).

Available from CPAN. See the module manpage for more information.

### Apache::WAP::MailPeek—Demonstrate Use of WML Delivery

This is a simple module to demonstrate the use of delivery of WML with mod_perl. It was written to accompany an article in the *Perl Journal* (Issue number 20).

Available from CPAN. See the module manpage for more information.

## Apache::Archive—Expose Archive Files Through the Apache Web Server

`Apache::Archive` is a mod_perl extension that allows the Apache HTTP server to expose *.tar* and *.tar.gz* archives on the fly. When a client requests such an archive file, the server will return a page displaying information about the file that allows the user to view or download individual files from within the archive.

Available from CPAN. See the module manpage for more information.

## Apache::Gateway—Implement a Gateway

The `Apache::Gateway` module implements a gateway using LWP with assorted optional features. From the HTTP/1.1 draft, a gateway is:

> `[a] server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.`

Features:

- Standard gateway features implemented using LWP
- Automatic failover with mirrored instances
- Multiplexing
- Pattern-dependent gatewaying
- FTP directory gatewaying
- Timestamp correction

Available from CPAN. See the module manpage for more information.

## Apache::NNTPGateway—NNTP Interface for a mod_perl-Enabled Apache Web Server.

Available from CPAN. See the module manpage for more information.

## Apache::PrettyPerl—Syntax Highlighting for Perl Files

An Apache mod_perl `PerlHandler` that outputs color syntax-highlighted Perl files in the client's browser.

Available from CPAN. See the module manpage for more information.

## Apache::PrettyText—Reformat .txt Files for Client Display

Dynamically formats *.txt* files so they look nicer in the client's browser.

Available from CPAN. See the module manpage for more information.

## Apache::RandomLocation—Random File Display

Given a list of locations in `ConfigFile`, this module will instruct the browser to redirect to one of them. The locations in `ConfigFile` are listed one per line, with lines beginning with # being ignored. How the redirection is handled depends on the variable `Type`.

Available from CPAN. See the module manpage for more information.

## Apache::Stage—Manage a Staging Directory

A staging directory is a place where the author of an HTML document checks the look and feel of the document before it's uploaded to the final location. A staging place doesn't need to be a separate server or a mirror of the "real" tree, or even a tree of symbolic links. A sparse directory tree that holds nothing but the staged files will do.

`Apache::Stage` implements a staging directory that needs a minimum of space. By default, the path for the per-user staging directory is hardcoded as:

```
/STAGE/any-user-name
```

The code respects proper internal and external redirects for any documents that are not in the staging directory tree. This means that all graphics are displayed as they will be when the staged files have been published. The following table provides an example structure:

```
Location           Redirect-to Comment
------------------ ----------- --------------------------
/STAGE/u1/         /           Homepage. Internal Redirect.
/STAGE/u2/dir1     /dir1/      Really /dir1/index.html
/STAGE/u3/dir2     /dir2/      Directory has no index.html
                               Options Indexes is off, thus
                               "Forbidden"
/STAGE/u4/dir2/foo /dir2/foo   Internal redirect.
/STAGE/u5/bar      -           Exists really, no redirect
                               necessary
/STAGE/u6          -           Fails unless location exists
```

The entries described in *SYNOPSIS* in *access.conf* or an equivalent place define the name of the staging directory, the name of an internal location that catches the exception when a document is not in the staging directory, and the regular expression that transforms the staging URI into the corresponding public URI.

With this setup only `ErrorDocuments` 403 and 404 will be served by `Apache::Stage`. If you need coexistence with different `ErrorDocument` handlers, you will either have to disable them for */STAGE* or integrate the code of `Apache::Stage` into an `if/else` branch based on the path.

Available from CPAN. See the module manpage for more information.

### Apache::Roaming—A mod_perl Handler for Roaming Profiles

With `Apache::Roaming` you can use your Apache web server as a Netscape Roaming Access server. This allows users to store Netscape Communicator 4.5+ preferences, bookmarks, address books, cookies, etc., on the server so that they can use (and update) the same settings from any Netscape Communicator 4.5+ browser that can access the server.

Available from CPAN. See the module manpage for more information.

### Apache::Backhand—Write mod_backhand Functions in Perl

`Apache::Backhand` ties mod_perl together with mod_backhand, in two major ways. First, the `Apache::Backhand` module itself provides access to the global and shared state information provided by mod_backhand (most notably server stats). Second, the `byPerl` C function (which is not part of the `Apache::Backhand` module but is distributed with it) allows you to write candidacy functions in Perl.

Available from CPAN. See the module manpage for more information.

## Database Modules

### Apache::DBI—Initiate a Persistent Database Connection

Covered in Chapter 20.

Available from CPAN. See the module manpage for more information.

### Apache::OWA—Oracle's PL/SQL Web Toolkit for Apache

This module makes it possible to run scripts written using Oracle's PL/SQL Web Toolkit under Apache.

Available from CPAN. See the module manpage for more information.

### Apache::Sybase::CTlib—Persistent CTlib Connection Management for Apache

Available from CPAN. See the module manpage for more information.

# Toolkits and Frameworks for Content-Generation and Other Phases

## Apache::ASP—Active Server Pages for Apache with mod_perl

`Apache::ASP` provides an Active Server Pages port to the Apache web server with Perl scripting *only* and enables developing of dynamic web applications with session management and embedded Perl code. There are also many powerful extensions, including XML taglibs, XSLT rendering, and new events not originally part of the ASP API.

Available from CPAN. See the module manpage for more information.

## Apache::AxKit—XML Toolkit for mod_perl

AxKit is a suite of tools for the Apache *httpd* server running mod_perl. It provides developers with extremely flexible options for delivering XML to all kinds of browsers, from hand-held systems to Braille readers to ordinary browsers. All this can be achieved using nothing but W3C standards, although the plug-in architecture provides the hooks for developers to write their own stylesheet systems, should they so desire. Two non-W3C stylesheet systems are included as examples.

The toolkit provides intelligent caching, which ensures that if any parameters in the display of the XML file change, the cache is overwritten. The toolkit also provides hooks for DOM-based stylesheets to cascade. This allows (for example) the initial stylesheet to provide menu items and a table of contents, while the final stylesheet formats the finished file to the desired look. It's also possible to provide multiple language support this way.

AxKit and its documentation are available from *http://www.axkit.org/*.

## HTML::Embperl—Embed Perl into HTML

`Embperl` gives you the power to embed Perl code in your HTML documents and the ability to build your web site out of small, reusable objects in an object-oriented style. You can also take advantage of all the standard Perl modules (including `DBI` for database access) and use their functionality to easily include their output in your web pages.

`Embperl` has several features that are especially useful for creating HTML, including dynamic tables, form-field processing, URL escaping/unescaping, session handling, and more.

Embperl is a server-side tool, which means that it's browser-independent. It can run in various ways: under mod_perl, as a CGI script, or offline.

For database access, there is a module called DBIx::Recordset that works well with Embperl and simplifies creating web pages with database content.

Available from CPAN. See the module manpage for more information.

## Apache::EmbperlChain—Process Embedded Perl in HTML in the OutputChain

Uses Apache::OutputChain to filter the output of content generators through Apache::Embperl.

Available from CPAN. See the module manpage for more information.

## Apache::ePerl—Embedded Perl 5 Language

ePerl interprets an ASCII file that contains Perl program statements by replacing any Perl code it finds with the result of evaluating that code (which may be chunks of HTML, or could be nothing) and passing through the plain ASCII text untouched. It can be used in various ways: as a standalone Unix filter or as an integrated Perl module for general file-generation tasks and as a powerful web-server scripting language for dynamic HTML page programming.

Available from CPAN. See the module manpage for more information.

## Apache::iNcom—E-Commerce Framework

Apache::iNcom is an e-commerce framework. It is not a ready-to-run merchant system. It integrates the different components needed for e-commerce into a coherent whole.

The primary design goals of the framework are flexibility and security. Most merchant systems will make assumptions about the structure of your catalog data and your customer data, or about how your order process works. Most also impose severe restrictions on how the programmer will interface with your electronic catalog. These are precisely the kinds of constraints that Apache::iNcom is designed to avoid.

Apache::iNcom provides the following infrastructure:

- Session management
- Cart management
- Input validation
- Order management

- User management
- Easy database access
- Internationalization
- Error handling

Most of the base functionality of `Apache::iNcom` is realized by using standard well-known modules such as `DBI` for generic SQL database access, `HTML::Embperl` for dynamic page generation, `Apache::Session` for session management, mod_perl for Apache integration, and `Locale::Maketext` for localization.

Here are its assumptions:

- Data is held in a SQL database that supports transactions.
- The user interface is presented using HTML.
- Sessions are managed through cookies.

Available from CPAN. See the module manpage for more information.

## Apache::Mason—Perl-Based Web Site Development and Delivery System

`Apache::Mason` allows web pages and sites to be constructed from shared, reusable building blocks called *components*. Components contain a mixture of Perl and HTML and can call each other and pass values back and forth like subroutines. Components increase modularity and eliminate repetitive work: common design elements (headers, footers, menus, logos) can be extracted into their own components, so that they need be changed only once to affect the whole site.

Other `Mason` features include powerful filtering and templating facilities, an HTML/data-caching model, and a web-based site-previewing utility.

Available from CPAN and *http://www.masonhq.com/*. See the module manpage for more information.

## Apache::PageKit—Web Applications Framework

`Apache::PageKit` is a web applications framework that is based on mod_perl. This framework is distinguished from others (such as `Embperl` and `Mason`) by providing a clear separation of programming, content, and presentation. It does this by implementing a Model/View/Content/Controller (MVCC) design paradigm:

- Model is implemented by user-supplied Perl classes
- View is a set of HTML templates
- Content is a set of XML files
- Controller is `PageKit`

This allows programmers, designers, and content editors to work independently, using clean, well-defined interfaces.

`Apache::PageKit` provides the following features:

- Component-based architecture
- Language localization
- Session management
- Input validation
- Sticky HTML forms
- Authentication
- Co-branding
- Automatic dispatching of URIs
- Easy error handling

Available from CPAN. See the module manpage for more information.

### Template Toolkit—Template Processing System

The Template Toolkit is a collection of modules that implements a fast, flexible, powerful, and extensible template processing system. It was originally designed for generating dynamic web content, but it can be used equally well for processing any other kind of text-based documents (HTML, XML, POD, PostScript, LaTeX, etc.).

It can be used as a standalone Perl module or embedded within an Apache/mod_perl server for generating highly configurable dynamic web content. A number of Perl scripts are also provided that can greatly simplify the process of creating and managing static web content and other offline document systems.

The `Apache::Template` module provides a simple mod_perl interface to the Template Toolkit.

Available from CPAN. It's covered in Appendix D and at *http://tt2.org/*.

## Output Filters and Layering Modules

### Apache::OutputChain—Chain Stacked Perl Handlers

`Apache::OutputChain` was written to explore the possibilities of stacked handlers in mod_perl. It ties `STDOUT` to an object that catches the output and makes it easy to build a chain of modules that work on the output data stream.

Examples of modules that are built using this idea are `Apache::SSIChain`, `Apache::GzipChain`, and `Apache::EmbperlChain`—the first processes the SSIs in the stream, the second compresses the output on the fly, and the last provides `Embperl` processing.

The syntax is like this:

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

The modules are listed in *reverse* order of their execution—here the `Apache::PassHtml` module simply collects a file's content and sends it to `STDOUT`, and then it's processed by `Apache::SSIChain`, which sends its output to `STDOUT` again. Then it's processed by `Apache::OutputChain`, which finally sends the result to the browser.

An alternative to this approach is `Apache::Filter`, which has a more natural *forward* configuration order and is easier to interface with other modules.

`Apache::OutputChain` works with `Apache::Registry` as well. For example:

```
Alias /foo /home/httpd/perl/foo
<Location /foo>
    SetHandler "perl-script"
    Options +ExecCGI
    PerlHandler Apache::OutputChain Apache::GzipChain Apache::Registry
</Location>
```

It's really a regular `Apache::Registry` setup, except for the added modules in the `PerlHandler` line.

Available from CPAN. See the module manpage for more information.

## Apache::Clean—mod_perl Interface Into HTML::Clean

`Apache::Clean` uses `HTML::Clean` to tidy up large, messy HTML, saving bandwidth. It is particularly useful with `Apache::Compress` for maximum size reduction.

Available from CPAN. See the module manpage for more information.

## Apache::Filter—Alter the Output of Previous Handlers

In the following configuration:

```
<Files ~ "*\.fltr">
    SetHandler perl-script
    PerlSetVar Filter On
    PerlHandler Filter1 Filter2 Filter3
</Files>
```

each of the handlers `Filter1`, `Filter2`, and `Filter3` will make a call to `$r->filter_input()`, which will return a file handle. For `Filter1`, the file handle points to the requested file. For `Filter2`, the file handle contains whatever `Filter1` wrote to `STDOUT`. For `Filter3`, it contains whatever `Filter2` wrote to `STDOUT`. The output of `Filter3` goes directly to the browser.

Available from CPAN. See the module manpage for more information.

## Apache::GzipChain—Compress HTML (or Anything) in the OutputChain

Covered in Chapter 13.

Available from CPAN. See the module manpage for more information.

## Apache::PassFile—Send File via OutputChain

See `Apache::GzipChain`. It's a part of the same package as `Apache::GzipChain`.

## Apache::Gzip—Auto-Compress Web Files with gzip

Similar to `Apache::GzipChain` but works with `Apache::Filter`.

This configuration:

```
PerlModule Apache::Filter
<Files ~ "*\.html">
    SetHandler perl-script
    PerlSetVar Filter On
    PerlHandler Apache::Gzip
</Files>
```

will send all the **.html* files compressed if the client accepts the compressed input.

And this one:

```
PerlModule Apache::Filter
Alias /home/http/perl /perl
<Location /perl>
    SetHandler perl-script
    PerlSetVar Filter On
    PerlHandler Apache::RegistryFilter Apache::Gzip
</Location>
```

will compess the output of the `Apache::Registry` scripts. Note that you should use `Apache::RegistryFilter` instead of `Apache::Registry` for this to work.

You can use as many filters as you want:

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

You can test that it works by either looking at the size of the response in the *access.log* file or by telnet:

```
panic% telnet localhost 8000
Trying 127.0.0.1
Connected to 127.0.0.1
```

```
Escape character is '^]'.
GET /perl/test.pl HTTP 1.1
Accept-Encoding: gzip
User-Agent: Mozilla
```

You will get the data compressed if it's configured correctly.

## Apache::Compress—Auto-Compress Web Files with gzip

This module lets you send the content of an HTTP response as *gzip*-compressed data. Certain browsers (e.g., Netscape and IE) can request content compression via the `Content-Encoding` header. This can speed things up if you're sending large files to your users through slow connections.

Browsers that don't request *gzip*ped data will receive uncompressed data.

This module is compatibile with `Apache::Filter`, so you can compress the output of other content generators.

Available from CPAN. See the module manpage for more information.

## Apache::Layer—Layer Content Tree Over One or More Others

This module is designed to allow multiple content trees to be layered on top of each other within the Apache server.

Available from CPAN. See the module manpage for more information.

## Apache::Sandwich—Layered Document (Sandwich) Maker

The `Apache::Sandwich` module allows you to add per-directory custom "header" and "footer" content to a given URI. Works only with `GET` requests. Output of combined parts is forced to *text/html*. The handler for the sandwiched document is specified by the `SandwichHandler` configuration variable. If it is not set, `default-handler` is used.

The basic concept is that the concatenation of the header and footer parts with the sandwiched file in between constitutes a complete valid HTML document.

Available from CPAN. See the module manpage for more information.

## Apache::SimpleReplace—Simple Template Framework

`Apache::SimpleReplace` provides a simple way to insert content within an established template for uniform content delivery. While the end result is similar to `Apache::Sandwich`, `Apache::SimpleReplace` offers two main advantages:

- It does not use separate header and footer files, easing the pain of maintaining syntactically correct HTML in separate files.

- It is `Apache::Filter` aware, so it can both accept content from other content handlers and pass its changes on to others later in the chain.

Available from CPAN. See the module manpage for more information.

### Apache::SSI—Implement Server-Side Includes in Perl

`Apache::SSI` implements the functionality of mod_include for handling server-parsed HTML documents. It runs under Apache's mod_perl.

There are two main reasons you might want to use this module: you can subclass it to implement your own custom SSI directives, and you can parse the output of other mod_perl handlers or send the SSI output through another handler (use `Apache::Filter` to do this).

Available from CPAN. See the module manpage for more information.

## Logging-Phase Handlers

### Apache::RedirectLogFix—Correct Status While Logging

Because of the way mod_perl handles redirects, the status code is not properly logged. The `Apache::RedirectLogFix` module works around this bug until mod_perl can deal with this. All you have to do is to enable it in the *httpd.conf* file.

```
PerlLogHandler Apache::RedirectLogFix
```

For example, you will have to use it when doing:

```
$r->status(304);
```

and do some manual header sending, like this:

```
$r->status(304);
$r->send_http_header();
```

Available from the mod_perl distribution. See the module manpage for more information.

### Apache::DBILogConfig—Logs Access Information in a DBI Database

This module replicates the functionality of the standard Apache module mod_log_config but logs information in a DBI-compatible database instead of a file.

Available from CPAN. See the module manpage for more information.

### Apache::DBILogger—Tracks What's Being Transferred in a DBI Database

This module tracks what's being transferred by the Apache web server in SQL database (everything with a DBI/DBD driver). This allows you to get statistics (of almost everything) without having to parse the log files (as with the `Apache::Traffic` module, but using a "real" database, and with a lot more logged information).

After installation, follow the instructions in the synopsis and restart the server. The statistics are then available in the database.

Available from CPAN. See the module manpage for more information.

### Apache::DumpHeaders—Watch HTTP Transaction via Headers

This module is used to watch an HTTP transaction, looking at the client and server headers. With `Apache::ProxyPassThru` configured, you can watch your browser talk to any server, not just the one that is using this module.

Available from CPAN. See the module manpage for more information.

### Apache::Traffic—Track Hits and Bytes Transferred on a Per-User Basis

This module tracks the total number of hits and bytes transferred per day by the Apache web server, on a per-user basis. This allows for real-time statistics without having to parse the log files.

After installation, add this to your server's *httpd.conf* file:

```
PerlLogHandler  Apache::Traffic
```

and restart the server. The statistics will then be available through the traffic script, which is included in the distribution.

Available from CPAN. See the module manpage for more information.

## Core Apache Modules

### Apache::Module—Interface to Apache C Module Structures

This module provides an interface to the list of Apache modules configured with your *httpd* server and their `module *` structures.

Available from CPAN. See the module manpage for more information.

## Apache::ShowRequest—Show Phases and Module Participation

Part of the `Apache::Module` package. This module allows you to see the all phases of the request and what modules are participating in each of the phases.

Available from CPAN. See the module manpage for more information.

## Apache::SubProcess—Interface to Apache Subprocess API

The output of `system()`, `exec()`, and `open(PIPE,"|program")` calls will not be sent to the browser unless your Perl interpreter was configured with `sfio`.

One workaround is to use backticks:

```
print `command here`;
```

But a cleaner solution is provided by the `Apache::SubProcess` module. It overrides the `exec()` and `system()` calls with calls that work correctly under mod_perl.

Let's look at a few examples. This example overrides the built-in `system()` function and sends the output to the browser:

```
use Apache::SubProcess qw(system);
my $r = shift;
$r->send_http_header('text/plain');

system "/bin/echo hi there";
```

This example overrides the built-in `exec()` function and sends the output to the browser. As you can guess, the `print` statement after the `exec()` call will never be executed.

```
use Apache::SubProcess qw(exec);
my $r = shift;
$r->send_http_header('text/plain');

exec "/usr/bin/cal";

print "NOT REACHED\n";
```

The `env()` function sets an environment variable that can be seen by the main process and subprocesses, then it executes the */bin/env* program via `call_exec()`. The main code spawns a process, and tells it to execute the `env()` function. This call returns an output file handle from the spawned child process. Finally, it takes the output generated by the child process and sends it to the browser via `send_fd()`, which expects the file handle as an argument:

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');
```

```
my $efh = $r->spawn_child(\&env);
$r->send_fd($efh);

sub env {
    my $fh = shift;
    $fh->subprocess_env(HELLO => 'world');
    $fh->filename("/bin/env");
    $fh->call_exec;
}
```

This example is very similar to the previous example, but it shows how you can pass arguments to the external process. It passes the string to print as a banner via a subprocess:

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

my $fh = $r->spawn_child(\&banner);
$r->send_fd($fh);

sub banner {
    my $fh = shift;
    # /usr/games/banner on many Unices
    $fh->filename("/usr/bin/banner");
    $fh->args("-w40+Hello%20World");
    $fh->call_exec;
}
```

The last example shows how you can have full access to the STDIN, STDOUT, and STDERR streams of the spawned subprocess, so that you can pipe data to a program and send its output to the browser:

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

use vars qw($string);
$string = "hello world";
my($out, $in, $err) = $r->spawn_child(\&echo);
print $out $string;
$r->send_fd($in);

sub echo {
    my $fh = shift;
    $fh->subprocess_env(CONTENT_LENGTH => length $string);
    $fh->filename("/tmp/pecho");
    $fh->call_exec;
}
```

The echo( ) function is similar to the earlier example's env( ) function. */tmp/pecho* is as follows:

```
#!/usr/bin/perl
read STDIN, $buf, $ENV{CONTENT_LENGTH};
print "STDIN: '$buf' ($ENV{CONTENT_LENGTH})\n";
```

In the last example, a string is defined as a global variable, so its length could be calculated in the `echo( )` function. The subprocess reads from `STDIN`, to which the main process writes the string ("hello world"). It reads only the number of bytes specified by the `CONTENT_LENGTH` environment variable. Finally, the external program prints the data that it read to `STDOUT`, and the main program intercepts it and sends it to the client's socket (i.e., to the browser).

This module is also discussed in Chapter 10.

Available from CPAN. See the module manpage for more information.

## Apache::Connection—Interface to the Apache conn_rec Data Structure

This module provides the Perl interface to the `conn_rec` data structure, which includes various records unique to each connection, such as the state of a connection, server and base server records, child number, etc. See *include/httpd.h* for a complete description of this data structure.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::Constants—Constants Defined in httpd.h

Server constants (`OK`, `DENIED`, `NOT_FOUND`, etc.) used by Apache modules are defined in *httpd.h* and other header files. This module gives Perl access to those constants.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::ExtUtils—Utilities for Apache C/Perl Glue

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::File—Advanced Functions for Manipulating Files on the Server Side

`Apache::File` does two things. First, it provides an object-oriented interface to file handles, similar to Perl's standard `IO::File` class. While the `Apache::File` module does not provide all the functionality of `IO::File`, its methods are approximately twice as fast as the equivalent `IO::File` methods. Secondly, when you use `Apache::File`, it adds to the Apache class several new methods that provide support for handling files under the HTTP/1.1 protocol.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::Log—Interface to Apache Logging

The `Apache::Log` module provides an interface to Apache's `ap_log_error()` and `ap_log_rerror()` routines.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::LogFile—Interface to Apache's Logging Routines

The `PerlLogFile` directive from this package can be used to hook a Perl file handle to a piped logger or to a file open for appending. If the first character of the filename is a "|", the file handle is opened as a pipe to the given program. The file or program can be relative to the `ServerRoot`.

So if *httpd.conf* contains these settings:

```
PerlModule Apache::LogFile
PerlLogFile |perl/mylogger.pl My::Logger
```

in your code you can log to the `My::Logger` file handle:

```
print My::Logger "a message to the Log"
```

and it'll be piped through the *perl/mylogger.pl* script.

Available from CPAN. See the module manpage for more information.

## Apache::Scoreboard—Perl Interface to Apache's scoreboard.h

Apache keeps track of server activity in a structure known as the scoreboard. There is a slot in the scoreboard for each child server, containing information such as status, access count, bytes served, and CPU time. This information is also used by mod_status to provide server statistics in a human-readable form.

Available from CPAN. See the module manpage for more information.

## Apache::Server—Perl Interface to the Apache server_rec Struct

The `Apache::Server` class contains information about the server's configuration. Using this class it's possible to retrieve any data set in *httpd.conf* and `<Perl>` sections.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::Table—Perl Interface to the Apache Table Struct

This module provides tied interfaces to Apache data structures. By using it you can add, merge, and clear entries in *headers_in*, *headers_out*, *err_headers_out*, *notes*, *dir_config*, and *subprocess_env*.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::URI—URI Component Parsing and Unparsing

This module provides an interface to the Apache `util_uri` module and the `uri_components` structure. The available methods are: `parsed_uri( )`, `parse( )`, `unparse( )`, `scheme( )`, `hostinfo( )`, `user( )`, `password( )`, `hostname( )`, `port( )`, `path( )`, `rpath( )`, `query( )`, and `fragment( )`.

Supplied with the mod_perl distribution. See the module manpage for more information.

### Apache::Util—Perl Interface to Apache C Utility Functions

This module provides a Perl interface to some of the C utility functions available in Apache. The same functionality is avaliable in *libwww-perl*, but the C versions are faster: `escape_html( )`, `escape_uri( )`, `unescape_uri( )`, `unescape_uri_info( )`, `parsedate( )`, `ht_time( )`, `size_string( )`, and `validate_password( )`.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Other Miscellaneous Modules

### Apache::Session—Maintain Session State Across HTTP Requests

This module provides mod_perl with a mechanism for storing persistent user data in a global hash, which is independent of the underlying storage mechanism. Currently it supports storage in standard files, DBM files, or a relational database using `DBI`. Read the manpage of the mechanism you want to use for a complete reference.

`Apache::Session` provides persistence to a data structure. The data structure has an ID number, and you can retrieve it by using the ID number. In the case of Apache, you would store the ID number in a cookie or the URL to associate it with one browser, but how you handle the ID is completely up to you. The flow of things is generally:

```
    Tie a session to Apache::Session.
    Get the ID number.
    Store the ID number in a cookie.
    End of Request 1.

    (time passes)

    Get the cookie.
    Restore your hash using the ID number in the cookie.
    Use whatever data you put in the hash.
    End of Request 2.
```

Using Apache::Session is easy: simply tie a hash to the session object, put any data structure into the hash, and the data you put in automatically persists until the next invocation. Example B-1 is an example that uses cookies to track the user's session.

*Example B-1. session.pl*

```
# pull in the required packages
use Apache::Session::MySQL;
use Apache;

use strict;

# read in the cookie if this is an old session
my $r = Apache->request;
my $cookie = $r->header_in('Cookie');
$cookie =~ s/SESSION_ID=(\w+)/$1/;

# create a session object based on the cookie we got from the
# browser, or a new session if we got no cookie
my %session;
eval {
    tie %session, 'Apache::Session::MySQL', $cookie,
        {DataSource => 'dbi:mysql:sessions',
         UserName   => $db_user,
         Password   => $db_pass,
         LockDataSource => 'dbi:mysql:sessions',
         LockUserName   => $db_user,
         LockPassword   => $db_pass,
        };
};
if ($@) {
    # could be a database problem
    die "Couldn't tie session: $@";
}

# might be a new session, so let's give them their cookie back
my $session_cookie = "SESSION_ID=$session{_session_id};";
$r->header_out("Set-Cookie" => $session_cookie);
```

After %session is tied, you can put anything but file handles and code references into $session{_session_id};, and it will still be there when the user invokes the next page.

It is possible to write an Apache authentication handler using Apache::Session. You can put your authentication token into the session. When a user invokes a page, you open his session, check to see if he has a valid token, and authenticate or forbid based on that.

An alternative to Apache::Session is Apache::ASP, which has session-tracking abilities. HTML::Embperl hooks into Apache::Session for you.

Available from CPAN. See the module manpage for more information.

## Apache::RequestNotes—Easy, Consistent Access to Cookie and Form Data Across Each Request Phase

Apache::RequestNotes provides a simple interface allowing all phases of the request cycle access to cookie or form input parameters in a consistent manner. Behind the scenes, it uses *libapreq* (Apache::Request) functions to parse request data and puts references to the data in pnotes( ).

Once the request is past the PerlInitHandler phase, all other phases can have access to form input and cookie data without parsing it themselves. This relieves some strain, especially when the GET or POST data is required by numerous handlers along the way.

Available from CPAN. See the module manpage for more information.

## Apache::Cookie—HTTP Cookies Class

The Apache::Cookie module is a Perl interface to the cookie routines in *libapreq*. The interface is based on the CGI::Cookie module.

Available from CPAN. See the module manpage for more information.

## Apache::Icon—Look Up Icon Images

This module rips out the icon guts of mod_autoindex and provides a Perl interface for looking up icon images. The motivation is to piggy-back the existing AddIcon and related directives for mapping file extensions and names to icons, while keeping things as small and fast as mod_autoindex does.

Available from CPAN. See the module manpage for more information.

## Apache::Include—Utilities for mod_perl/mod_include Integration

The `Apache::Include` module provides a handler, making it simple to include `Apache::Registry` scripts with the mod_include Perl directive.

`Apache::Registry` scripts can also be used in mod_include-parsed documents using a *virtual include*.

The `virtual()` method may be called to include the output of a given URI in your Perl scripts. For example:

```
use Apache::Include ();
print "Content-type: text/html\n\n";

print "before include\n";
my $uri = "/perl/env.pl";
Apache::Include->virtual($uri);
print "after include\n";
```

The output of the perl CGI script located at */perl/env.pl* will be inserted between the "before include" and "after include" strings and printed to the client.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::Language—Perl Transparent Language Support for Apache Modules and mod_perl Scripts

The goal of this module is to provide a simple way for mod_perl module writers to include support for multiple language requests.

An `Apache::Language` object acts like a language-aware hash. It stores key/language/value triplets. Using the `Accept-Language` header field sent by the web client, it can choose the most appropriate language for the client. Its usage is transparent to the client.

Available from CPAN. See the module manpage for more information.

## Apache::Mmap—Perl Interface to the mmap(2) System Call

The `Apache::Mmap` module lets you use `mmap` to map in a file as a Perl variable rather than reading the file into dynamically allocated memory. It works only if your OS supports Unix or POSIX.1b `mmap()`. `Apache::Mmap` can be used just like `Mmap` under mod_perl.

Available from CPAN. See the module manpage for more information.

## Apache::GD::Graph—Generate Graphs in an Apache Handler

The primary purpose of this module is to provide a very easy-to-use, lightweight, and fast charting capability for static pages, dynamic pages, and CGI scripts, with the chart-creation process abstracted and placed on any server.

Available from CPAN. See the module manpage for more information.

## Apache::Motd—Provide motd (Message of the Day) Functionality to a Web Server

This module provides an alternative and more efficient method of notifying your web users of potential downtime or problems affecting your web server and web services.

Available from CPAN. See the module manpage for more information.

## Apache::ParseLog—Object-Oriented Perl Extension for Parsing Apache Log Files

`Apache::ParseLog` provides an easy way to parse the Apache log files, using object-oriented constructs. The module is flexible, and the data it generates can be used for your own applications (CGI scripts, simple text-only report generators, feeding an RDBMS, data for Perl/Tk-based GUI applications, etc.).

Available from CPAN. See the module manpage for more information.

## Apache::RegistryLoader—Compile Apache::Registry Scripts at Server Startup

Covered in Chapter 13.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::SIG—Override Apache Signal Handlers with Perl's Signal Handlers

Covered in Chapter 6.

Supplied with the mod_perl distribution. See the module manpage for more information.

## Apache::TempFile—Allocate Temporary Filenames for the Duration of a Request

This module provides unique paths for temporary files and ensures that they are removed when the current request is completed.

Available from CPAN. See the module manpage for more information.

## Xmms—Perl Interface to the xmms Media Player

A collection of Perl interfaces for the *xmms* media player. Includes a module that allows you to control *xmms* from the browser. mod_perl generates a page with an index of available MP3 files and control buttons. You click on the links and *xmms* plays the files for you.

Available from CPAN. See the module manpage for more information.

## Module::Use—Log and Load Used Perl Modules

`Module::Use` records the modules used over the course of the Perl interpreter's lifetime. If the logging module is able, the old logs are read and frequently used modules are loaded automatically.

For example, if configured as:

```
<Perl>
    use Module::Use (Counting, Logger => "Debug");
</Perl>

PerlChildExitHandler Module::Use
```

it will record the used modules only when the child exists, logging everything (debug level).