PART II
The mod_perl API

At its simplest, mod_perl is merely a way to add real performance to the CGI environment, and many programmers are content to leave it at that. However, if you read Chapter 2 and stood back a moment, the implications of it ought to give you pause. Yes, you can potentially configure Apache with a single `PerlModule` directive. Yes, you can have persistent database connections without changing a single line of your existing code. Yes, all of it in Perl. Staggering.

The remainder of this book discusses something even better—with mod_perl you can access the full functionality of Apache using Perl! Prior to mod_perl, the only way to create extensions to Apache was to write them in C, which came with the usual headaches of slow development lifecycles, memory management, lack of native string support, clunky regular expressions, and so on. For whatever drawbacks there are to writing extensions using C, though, the fact that Apache has the underlying ability to allow you to tie into things such as URI to filename mapping, authentication, and MIME type translation makes it perfect for creating robust, scalable, and enterprise-ready Web applications. Now, with mod_perl, Perl developers can leverage this incredible infrastructure. In fact, as you will find in the following pages, they can do much more.

Unlike the CGI environment, which is confined to content generation, it is possible for Perl code to run during any of Apache's operational phases:

- Parsing of the configuration file
- Initializing a child process
- After reading the headers
- URI to filename translation
- Merging of configuration directives

- First entry into a container directive
- Checking host-based access control
- Checking user credentials
- Verifying a user against a specific resource
- Determining the MIME type
- Fixing up the headers prior to a response
- Generating the actual content
- Logging the request
- Cleaning up afterward
- Shutting down a child
- Restarting the server

If fully configured, your mod_perl perl server will allow Perl code to be executed for any of these phases.

In the typical Apache server programming environment, the term handler is used to refer to the code that processes the content generation phase only. However, mod_perl takes this a step further and applies the term to its hooks into all the phases of the Apache lifecycle. In mod_perl, a module used by one or more of the previously mentioned phases is called a *handler*, and you will hear this term used frequently throughout the book. For the moment, understanding a mod_perl handler as a single subroutine named `handler()` contained within an ordinary Perl module is sufficient. For example

```
package My::Dinghy;

use strict;

sub handler {
   # do stuff here...
}
1;
```

For each of the Apache phases, configuring mod_perl to run one or more `handler()` subroutines is possible. mod_perl accomplishes this task by using the Apache C API to both register itself with each phase and provide a custom `httpd.conf` directive for

configuring the phase. With these custom directives, we can specify the appropriate mod_perl handler that will be responsible for processing the phase.

Here is a complete list of mod_perl handler interfaces into the Apache lifecycle, each corresponding to one of the operational phases previously discussed:

- `SERVER_CREATE()`, `DIR_CREATE()`, `DIR_MERGE()` subroutines
- `PerlChildInitHandler`
- `PerlPostReadRequestHandler`
- `PerlTransHandler`
- `DIR_MERGE()` subroutine
- `PerlHeaderParserHandler`
- `PerlAccessHandler`
- `PerlAuthenHandler` and `PerlAuthzHandler`
- `PerlTypeHandler`
- `PerlFixupHandler`
- `PerlHandler`
- `PerlLogHandler`
- `PerlCleanupHandler`
- `PerlChildExitHandler`
- `PerlRestartHandler`

Remember, your own Perl code can be called during any of these phases, dynamically and subtly modifying the operation of the server. The amount of power and flexibility this feature allows is stunning.

Before you can take full advantage of all that mod_perl has to offer, however, you need to understand some of the fundamental concepts that you will be expanding on in your own programming. Part II focuses on the mechanics of the mod_perl API, such as accessing the Apache request object, URI and file manipulation, and actually creating, tuning, and fully leveraging the power of handlers. Part III then explains how to use handlers within each phase of the Apache lifecycle to their full extent. Along the way, we hope to augment what you may already know with the experiences of others who struggled through mod_perl over the years.