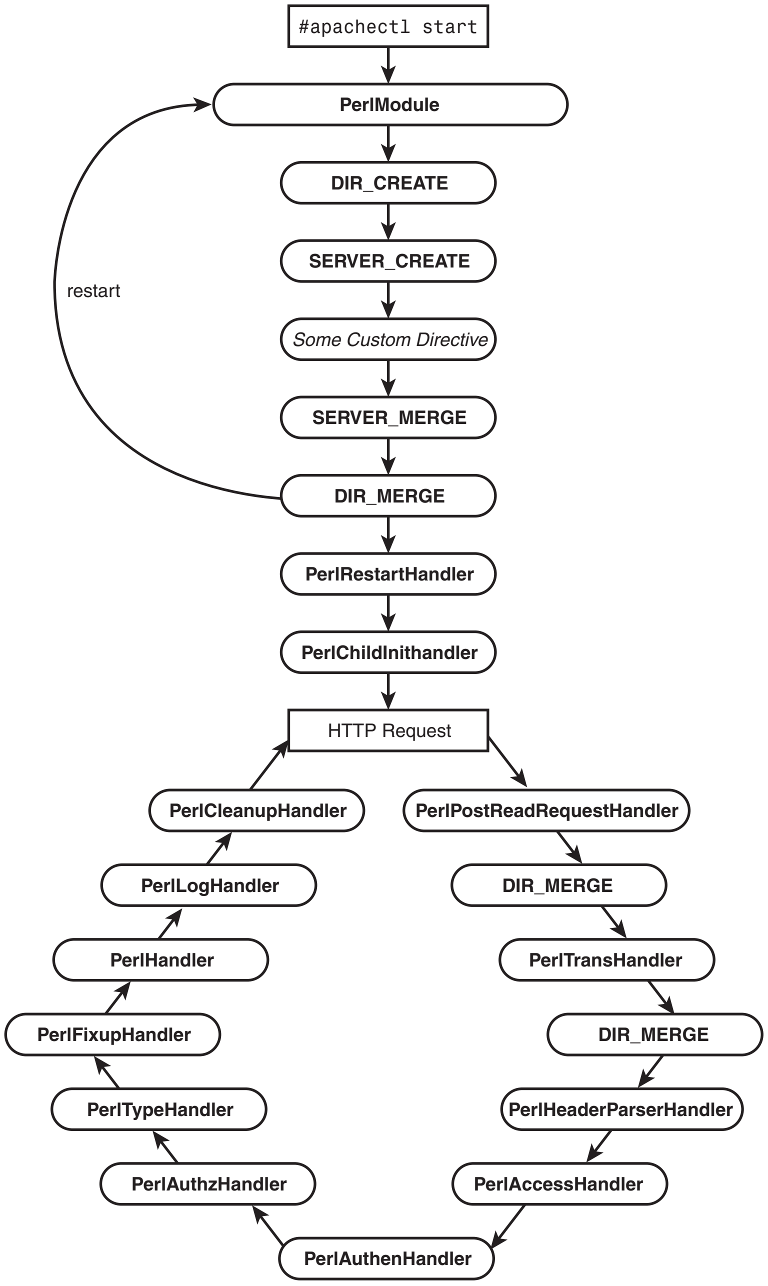
PART III
Programming the Apache Lifecycle

Now that you have been exposed to the mod_perl API, you might be scratching your head somewhat, asking "All this is nice, but now what do I do with it?" In order to leverage the full power of the Apache framework, you need to undergo a rather intense (and perhaps difficult) paradigm shift—just about *everything* about the way Apache works is now at your disposal and (potentially) under your control. Sometimes, knowing where to start is difficult.

This final part of the book explains the parts of the Apache lifecycle in detail: what the typical function of the phase is, what it is typically used for, and how you can mold it to your every whim in order to produce rather dramatic effects. Although we have touched on most of these phases to varying degrees, and you may already have a basic understanding of Apache's pre-fork architecture, now it is time to roll up our sleeves and get into the gory details.

To begin, this figure is an overview of the Apache (Unix) lifecycle from the point of view of the Perl module that contains your mod_perl handler.


#apachectl start
PerlModule
DIR_CREATE
SERVER_CREATE
Some Custom Directive
SERVER_MERGE
DIR_MERGE
restart
PerlRestartHandler
PerlChildInithandler
HTTP Request
PerlPostReadRequestHandler
DIR_MERGE
PerlTransHandler
DIR_MERGE
PerlHeaderParserHandler
PerlAccessHandler
PerlAuthenHandler
PerlAuthzHandler
PerlTypeHandler
PerlFixupHandler
PerlHandler
PerlLogHandler
PerlCleanupHandler

The first time your handler will get a chance to enter into the Apache lifecycle is when it is loaded with a `PerlModule` directive. This is the place where modules get to call any code they want to execute before any requests are processed: specifically, the code that exists in your module but outside of any subroutine. Recipe 8.3 shows an effective use of this initialization stage for creating a global shared memory cache. Keep in mind that, unlike the initializer hook provided to Apache C extension modules, handler initialization code will *not* be run when Apache is restarted unless you configure `PerlFreshRestart On`.

After your module is loaded and its initialization code run, things usually die down until request time. However, as we demonstrated in Recipes 7.8 and 7.10, mod_perl also offers the ability to enter into Apache's configuration creation and merge phases using directive handlers. The directive handler cycle is rather complex and the recipes in Chapter 7 that discussed it only really told part of the story. Now it is time to roll up our sleeves somewhat.

The first thing that happens when you implement a directive handler is that the per-server and per-directory entries for the module's namespace are created. mod_perl takes care of this behind the scenes when the module is loaded and before any of your module's custom configuration directives are seen. This is one of the reasons why you need to use the `PerlModule` directive to load your module before any of your custom configuration directives.

Next, Apache parses the directive itself, at which point mod_perl steps in and claims responsibility for the directive. The actual implementation of the directive is passed off from mod_perl proper to the Perl module registered to handle the directive. The directive subroutine is entered, where it can access either the per-server or per-directory configuration object and store its data.

As Apache traverses `httpd.conf`, it creates per-server objects for the main server and for each virtual server where the custom directive exists. Apache also creates a per-directory object for each directory where the custom directive is configured, as well as for any place where a per-directory directive exists on a per-server level. In the case of mod_perl directive handlers, the `SERVER_CREATE()` and `DIR_CREATE()` routines are used for this purpose if defined. As a final step, the per-server and per-directory entries are merged using the `SERVER_MERGE()` and `DIR_MERGE()` routines and the configuration process is complete.

At this point, Apache tosses the configuration it just worked out and starts parsing the configuration file all over again. Although it sounds strange, there are historical reasons for this. It is mainly done to ensure that Apache (or, more correctly, modules loaded into Apache) can survive a restart, but also just in case Apache is started with a `-d` option that differs from the `ServerRoot` directive found in the configuration file itself. The upshot of this double initialization is that Apache is now considered to be restarting, so the next thing to happen is that the `PerlRestartHandler` is run, giving you a hook into server initialization.

Under the current pre-fork model, the Apache parent process does not actually process any requests but instead forks off a number of child `httpd` processes that serve the incoming requests. For each child process that is spawned, a `PerlChildInitHandler` is run, after which Apache is ready to receive and process requests.

When a client initiates a request, an Apache child process steps up and the request cycle is entered. The Apache request cycle consists of a number of different phases, many of which have distinct and easily distinguishable purposes. However, a few are not as intuitive as one would like. Furthermore, some phases run all configured handlers until no more remain, whereas others terminate the phase on the first hint of success. It is these differences that make the request cycle somewhat intimidating at first, but hopefully something that the recipes in this Part can clarify.

Each of the chapters in this final Part explains a distinct part of the request cycle. However, it makes sense to see how they interact as a whole so we can sprinkle in some explanation that may not be clear upon examination of the phases individually.

The first thing that happens when Apache receives a request is that it parses the incoming HTTP message: the request line, incoming headers, and message body. Each of these parts is placed into the Apache request record where it can be accessed via the API during the phases to follow.

After the request record is created and populated, Apache begins to run the various phases of the request cycle. The first chance a Perl handler gets to operate on the request is with a `PerlPostReadRequestHandler`, where you can pre-process the request before any other phase gets the chance to see the incoming URI. After this initial chance for processing, the URI enters into the filename translation phase. Believe it or not, you can actually control the way that Apache maps the incoming URI to a physical file on the filesystem by installing a `PerlTransHandler`, which is sometimes quite a handy thing to be able to do.

After URI translation is complete, Apache knows to which `<Directory>`, `<Location>`, or similar container the request belongs. At this point, if Apache sees a custom directive within the container, it will run the `DIR_MERGE()` subroutine from your module to merge the configuration of the container with that of any parent (or of any per-directory configurations that reside on a per-server level). Depending on your configuration, you may see Apache call your directory merger both before and after URI translation, which is merely a result of how Apache handles the `<Location>` directive internally. This should not be of any great consequence, as long as you keep in mind that `DIR_MERGE()` can be called more than once per request.

Following URI translation and any per-directory merges, you are offered the ability to manipulate the request yet again using a `PerlHeaderParserHandler`. Although this phase is a bit of a misnomer (it has nothing to do with the actual parsing of the incoming headers), it actually was implemented within Apache prior to the post-read request phase, so the name persists for historical reasons. This is the first chance that you can operate on the request after the filename is known, and the first place to limit your interaction with requests filtered by `<Location>` and like directives.

Next comes the point where you get to control who is allowed to have access to your resources. This happens on three distinct levels. The `PerlAccessHandler` is for controlling access based on information contained at a server or connection level, while the `PerlAuthenHandler` grants resource access based on knowing the identity of the user. To get control at even a more granular level, the `PerlAuthzHandler` is there to restrict access based on attributes of authenticated users.

After the various access control phases are run, the requested resource is mapped to a MIME type using a `PerlTypeHandler`. In reality, this is probably the least-used phase of the request cycle, in part because the mod_mime C implementation is fast and efficient. Following MIME-type handling, you get one final chance to step in before you generate content using the `PerlFixupHandler`.

The `PerlHandler` is the real workhorse of the Apache request cycle, and it is here where you will spend most of your time, playing with the various templating modules and other cool features—content is king, after all. After laboring over the content to be sent to the client, the `PerlLogHandler` allows you to log the transaction and the `PerlCleanupHandler` to do any end-of-transaction processing.

Throughout each of these phases of the request cycle your handler needs to make decisions about the return value it will pass back to Apache—the value you choose can dramatically and drastically alter how Apache processes the remaining handlers for the request. If you recall from Recipe 3.12, Apache has two classes of response codes. The internal response codes, `OK`, `DECLINED`, and `DONE` indicate some measure of success. Anything else, such as `FORBIDDEN`, `REDIRECT` or `SERVER_ERROR` is considered to be an error from Apache's point of view. Returning an error code from a handler will force Apache into its error response cycle immediately, where it will process any configured `ErrorDocuments` or custom error responses.

For the Apache success codes, `OK`, `DECLINED`, and `DONE`, the path Apache takes is not as simple. Returning `DONE` from the Apache request cycle immediately sends the request to the logging phase. This is typically used to indicate that all content has been transmitted to the client and that no further handlers are required to run. For the other two return codes, `OK` and `DECLINED`, things are a bit more complex. For the `PerlTransHandler`, `PerlAuthenHandler`, `PerlAuthzHandler`, and `PerlTypeHandler`, the first handler to return `OK` ends the phase. For the remaining phases `OK` and `DECLINED` are essentially the same in that both allow other handlers in the same phase to run—choosing `OK` over `DECLINED` in these latter phases is more about writing self-documenting code than it is about the effect it will have on the other handlers.

Of course, over time Apache will terminate and spawn new children, so there will be a few extra `PerlChildInitHandlers` and `PerlChildExitHandlers` thrown into the mix, as well as the occasional `PerlRestartHandler` when you change configurations and restart Apache. But although features like directive handlers and `PerlChildInitHandlers` are nice tools to have, the bulk of your time will be spent programming and tweaking the various phases of the Apache request cycle. It is here that you gain access to the Apache request object and the majority of the methods and techniques discussed in earlier chapters. For this reason, the majority of this part is spent on examining the finer points of each of the phases of the Apache request cycle, though the final chapter does deal jointly with the remaining ancillary phases. Directive handlers are covered extensively in Chapter 7, and are included in many of the remaining examples.

Hopefully, within these final chapters you will find typical uses for all the request phases. Additionally, you will find some nonstandard uses that might pique your interest and send you in new directions. In either case, a more complete understanding of the Apache lifecycle should result, which will enable you to treat Apache more like an application server and less like a simple scripting engine.