

System Administration 3: Services

Lesson 1: **Getting Started**

[The Linux Learning Environment](#)

[Using the Linux Learning Environment](#)

[Logging In](#)

[Logging Out](#)

Lesson 2: **Services**

[Services: Behind the Scenes](#)

[Init and Runlevels](#)

[SysV versus Xinetd](#)

[Rc.local, Another Alternative](#)

Lesson 3: **BIND: Installation**

[The Berkeley Internet Name Daemon](#)

[Installing BIND](#)

[Initial Configuration](#)

Lesson 4: **BIND: Zone Files**

[Your First Zone File](#)

Lesson 5: **Building Software**

[Locating and Downloading the Source](#)

[Building Software From Source](#)

Lesson 6: **HTTPD: Introduction and Initial Configuration**

[An Introduction to Web Servers](#)

[Initial Configuration Changes](#)

Lesson 7: **HTTPD: Configuration**

[Enabling Userdir in Httpd](#)

[Per-Directory Access Controls](#)

Lesson 8: **Postfix: Introduction**

[An Introduction to Mail Servers](#)

[Getting Familiar With Postfix](#)

[Email Aliases](#)

Lesson 9: **Postfix: Security**

[An Introduction to Postfix Security](#)

[Controlling Relay Access](#)

[Greylisting: Another Method For Relay Access Control](#)

Lesson 10: **System Logs**

[What the Logs Tell You](#)

[Ways Of Viewing Log Files](#)

[Grep, More In Depth](#)

[An Introduction to Regular Expressions](#)

Lesson 11: **PHP**

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Getting Started

Welcome to the O'Reilly School of Technology (OST) System Administration 3 Course!

Course Objectives

When you complete this course, you will be able to:

- manage SysV- and inetd-based services.
- install and configure the BIND DNS server.
- create new DNS zones.
- build and install Apache's httpd server from a source package.
- configure Apache's httpd server using virtual hosts.
- use basic concepts related to HTTP.
- configure a Postfix-based email server.
- use basic SMTP concepts.
- install and configure PHP.

In this third course, you will delve into more advanced system administration topics and tasks, such as DNS, HTTP, and SMTP. Along with exploring these topics generally, you will install server software from both packages and source in order to implement these services on your own server.

From beginning to end, you will learn by doing real projects within the OST Learning Sandbox, including a unique environment that allows the student administrative access to their very own virtual server. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add **looks like this**.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type **look like this**.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

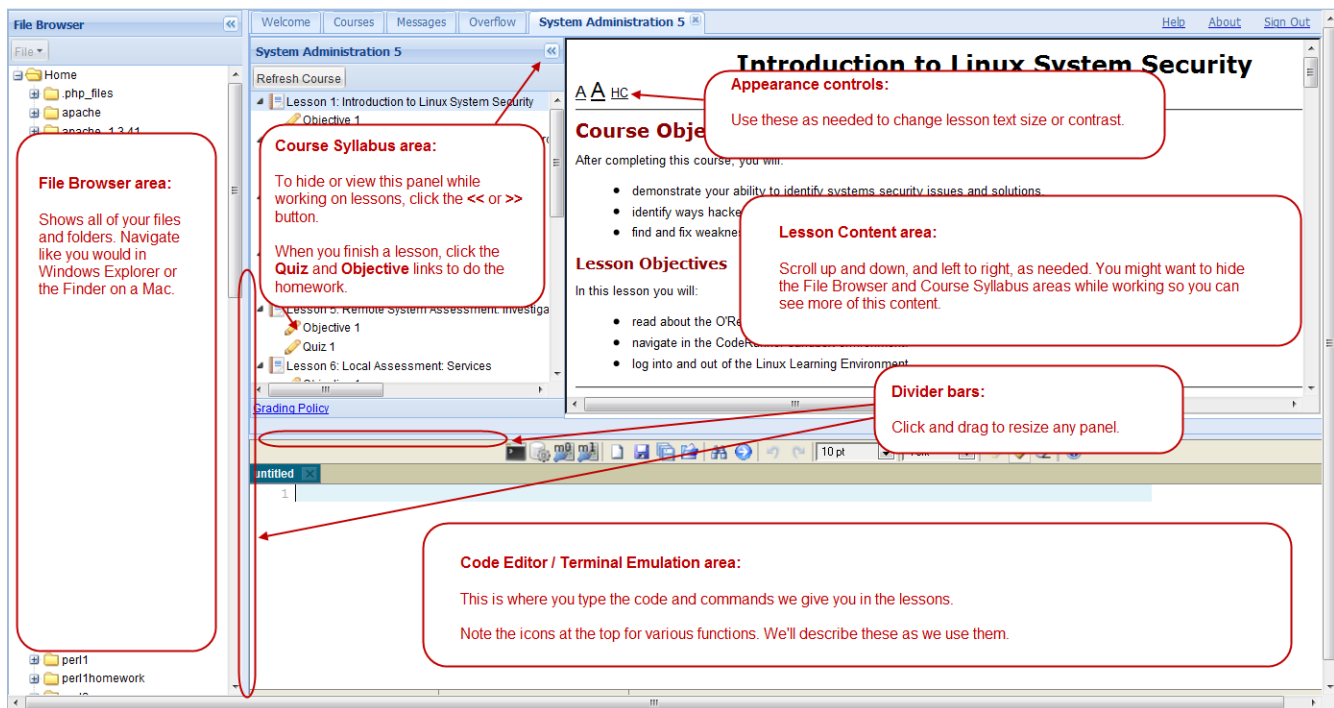
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

[File Management Demo](#)

[Code Editor Demo](#)

[Coursework Demo](#)

The Linux Learning Environment

For this course, we'll use CodeRunner to connect to and interact with the O'Reilly School of Technology *Linux Learning Environment*. The Linux Learning Environment gives you access to one or more Linux virtual machines to use as a tool to learn Linux. The Linux virtual machine that we'll be using is nearly identical to a physical machine in functionality. Each student receives his or her own virtual machine for this course in order to allow the student complete control over the system.

Using the Linux Learning Environment

In order to use the Linux Learning Environment, you must first connect to it. To do so, click on the **Linux Learning Environment** (🖥️) button located in the toolbar inside the CodeRunner window.

Logging In

The console messages from your machine's boot process print on the screen. Eventually you will see a login prompt like this:

```

OBSERVE:
CentOS Linux release 6.0 (Final)
Kernel 2.6.32.43 on x86_64

username-m0 login:

```

Type your username and press **Enter**, then type your password and press **Enter**. Your Linux virtual machine actually has four consoles, labeled tty0 through tty3, available for you to log in on. You can access these consoles by clicking the **left** (←) and **right** (→) arrow buttons on the toolbar. You can also use the buttons to navigate back to the Linux Learning Environment menu.

Note

You may be curious about why the consoles are called TTYs. This is for historical reasons. Early computers used devices called *teletypes* to allow users to interact with them. Teletypes generally had a keyboard for input and a printer that would print output from the computer so that the user could read it. The term *tty* is derived from the word **t**ele**t**ype.

Logging Out

It is generally a good idea to log out of your machine when you are finished working with it. In the real world, this keeps unauthorized people from using your account to harm the system. To log out of your Linux system, do the following:

CODE TO TYPE:

```
[username@username-m0 ~]$ exit
```

A new login prompt appears on the screen. This confirms that you have successfully logged out.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Services

In the previous course, we introduced to the concept of a service on your machine with the OpenSSH server. Here we will go more in depth, giving more explanation about how they are started and stopped, along with introducing a new tool, **chkconfig**.

Services: Behind the Scenes

On most Linux machines, there are two kinds of services, SysV services and *xinetd* services. All of the services that we'll cover in this course will be SysV-style services, but you'll probably encounter *xinetd* services in your career, so we'll outline the differences between the two. The primary difference between these two classes of services is how they are configured to start and stop. Before we get into that, though, we must discuss *runlevels*.

Init and Runlevels

In the first course, we briefly introduced **init** in the processes lesson. Init is the parent of all processes on the system, and it is the first process started by the kernel when the system starts to boot. Once started, it is init's job to bring up all the functions of the system at specified times in order to put the system into a particular state. These states are referred to as runlevels. When a particular runlevel is reached by init, services configured for that runlevel are started or stopped, depending on their configuration. For normal usage, init is configured to step through a particular set of runlevels, finally stopping at the runlevel that the system operates in normally. For your system, this is runlevel 3. Below are the six different runlevels on your system.

Runlevel	Description
0	Halt: Your machine enters this runlevel when it's shutting down.
1	Single-User Mode: Your system uses this runlevel for administrative tasks. A minimal set of services is started, and only one console login is allowed.
2	Multi-User Mode (no networking): The system operates normally and multiple users can log in on the console, but no networking services are started.
3	Multi-User Mode (with networking): The default runlevel.
4	Undefined: This runlevel isn't generally used, but it can be configured for special purposes.
5	Multi-User Mode (with GUI): Your system uses this runlevel when a graphical user interface is desired. Users can log in on both tty and graphical consoles.
6	Reboot: Your machine enters this runlevel when it's rebooting.

SysV-style services have scripts stored in a special directory on your machine, **/etc/init.d**. These are shell scripts, called *init scripts*, that handle the process of starting and stopping the service, as well as other functions we'll discuss later. In addition to this directory, there are several other directories, **/etc/rc0.d** through **/etc/rc6.d**. Each of these directories corresponds to one of the six runlevels. Symlinks are created inside these directories that point to the scripts in **/etc/init.d**, and init uses these symlinks to determine what services should be started or stopped in that runlevel. Let's look at the contents of **/etc/rc3.d**:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ls -l /etc/rc3.d/
total 0
lrwxrwxrwx 1 root root 19 Mar  7 14:59 K10sasauthd -> ../init.d/sasauthd
lrwxrwxrwx 1 root root 20 Mar  7 14:59 K50netconsole -> ../init.d/netconsole
lrwxrwxrwx 1 root root 14 Mar  7 14:59 K74ntpd -> ../init.d/ntpd
lrwxrwxrwx 1 root root 17 Mar  7 14:59 K75ntpd -> ../init.d/ntpd
lrwxrwxrwx 1 root root 15 Mar  7 14:59 K89rdisc -> ../init.d/rdisc
lrwxrwxrwx 1 root root 18 Mar 23 13:30 S08iptables -> ../init.d/iptables
lrwxrwxrwx 1 root root 17 Mar 23 13:30 S10network -> ../init.d/network
lrwxrwxrwx 1 root root 17 Mar  7 14:59 S12rsyslog -> ../init.d/rsyslog
lrwxrwxrwx 1 root root 15 Mar 23 13:30 S25netfs -> ../init.d/netfs
lrwxrwxrwx 1 root root 19 Mar  7 14:59 S26udev-post -> ../init.d/udev-post
lrwxrwxrwx 1 root root 15 Mar 23 13:30 S50setup -> ../init.d/setup
lrwxrwxrwx 1 root root 14 Mar 13 15:39 S55sshd -> ../init.d/sshd
lrwxrwxrwx 1 root root 16 Mar 19 16:04 S56xinetd -> ../init.d/xinetd
lrwxrwxrwx 1 root root 17 Mar  7 14:59 S80postfix -> ../init.d/postfix
lrwxrwxrwx 1 root root 15 Mar  7 14:59 S90crond -> ../init.d/crond
lrwxrwxrwx 1 root root 11 Mar 23 13:30 S99local -> ../rc.local
```

As you can see, the names of the symlinks follows a particular pattern, S or K, followed by a two-digit number, and then the service name. The S or K in the symlink name tells init whether to **Start** or **Kill** the service—that is, start the service if it's not already running or kill the service if it is running. The two-digit number gives a method to start or kill the services in a particular order, with the lowest-numbered services starting or stopping first. While it's possible to maintain these symlinks in each directory by hand (this is how it was done in the old days), there is a tool called **chkconfig** that makes this task much easier. Chkconfig can be used to display and edit what services start in what runlevels. First, we'll use chkconfig to see what services we have configured.

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ chkconfig --list
crond            0:off   1:off   2:on    3:on    4:on    5:on    6:off
iptables        0:off   1:off   2:on    3:on    4:on    5:on    6:off
netconsole      0:off   1:off   2:off   3:off   4:off   5:off   6:off
netfs           0:off   1:off   2:off   3:on    4:on    5:on    6:off
network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
ntpd            0:off   1:off   2:off   3:off   4:off   5:off   6:off
ntpddate        0:off   1:off   2:off   3:off   4:off   5:off   6:off
postfix         0:off   1:off   2:on    3:on    4:on    5:on    6:off
rdisc           0:off   1:off   2:off   3:off   4:off   5:off   6:off
rsyslog         0:off   1:off   2:on    3:on    4:on    5:on    6:off
sasauthd        0:off   1:off   2:off   3:off   4:off   5:off   6:off
setup           0:off   1:off   2:on    3:on    4:on    5:on    6:off
sshd            0:off   1:off   2:on    3:on    4:on    5:on    6:off
udev-post       0:off   1:on    2:on    3:on    4:on    5:on    6:off
xinetd          0:off   1:off   2:off   3:on    4:on    5:on    6:off

xinetd based services:
    telnet:      on
```

Services are listed one per line, and each column in the output tells us whether the service is set to start (on) or set to stop (off) in a particular runlevel. All six runlevels are listed. In addition to the SysV-style services, at the very bottom there is also a list of xinetd-based services. Interestingly enough, xinetd is itself a SysV-style service (we will discuss this more soon). To list a specific service rather than all of them, run **chkconfig --list service_name**. To set a services to start or stop, run **chkconfig service_name on** or **off**. Modern init scripts contain information in a header at the beginning of the script that describes dependencies for that service. **Chkconfig** uses this dependency information to determine what number to assign to the symlink in order to have the script run at the correct time in the runlevel.

SysV versus Xinetd

Now we can get into the differences between SysV- and xinetd-style services. As you saw before, xinetd is actually a SysV-style service, but it is responsible for managing its own kind of service. For this reason, it's often called a *super-server*. Much like other SysV-style services, xinetd runs in the background on your system and waits for some action to occur. In this case, it starts listening on various network sockets, depending on what services it's configured to provide. When a connection is started on that socket, xinetd starts the corresponding service and passes traffic between the network socket and the standard input and output streams of the service. This method is preferred for services that don't require high performance because it allows them to only run when necessary, saving memory and computing time. On the other hand, SysV-style services run constantly, and handle all of their own input and output, network or otherwise. For services like web and email, this gives a big edge on performance at the cost of resource usage. This is almost always an acceptable tradeoff, because the server's reason for existing is to provide that service.

Rc.local, Another Alternative

Sometimes you will want to add some functionality to the boot process, such as running a script, but you don't want to go through the process of writing a fully compliant SysV-style init script. Luckily, most distributions provide a method for doing this with the file `/etc/rc.local`. This file is a script that is executed at the end of runlevels 2-5, and you can add just about anything to it. As a test, go ahead and edit the file (you must do so as root, so use `sudo`) and add the [highlighted](#) line :

CODE TO TYPE:

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

touch /var/lock/subsys/local

echo "Hello, I'm init. I'm speaking to you from rc.local."
```

After you write the file out, you can test to see what your changes have done. Instead of rebooting your system using the "reboot" command, however, try using the command designed to switch runlevels on a running system, **telinit**. From the above table of runlevels, we know that the runlevel associated with rebooting the system is 6. This means you will want to run **telinit 6**. This operation requires root privileges as well. Immediately you'll begin to see the system rebooting. Eventually, you'll be presented with a login prompt when the system finishes its boot process. Right before that in the console you should see the message that you added to `rc.local` to echo:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo telinit 6...
Starting xinetd: [ OK ]
Starting postfix: [ OK ]
Starting crond: [ OK ]
Hello, I'm init. I'm speaking to you from rc.local.

CentOS release 6.2 (Final)
Kernel 2.6.32.43 on an x86_64

username-m0 login:
```

Now that you have seen what's happening behind the scenes when services start and stop, we can get into installing and configuring the services themselves. In the next lesson we'll install and configure the Berkeley Internet Name Daemon, the most widely used DNS server.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

BIND: Installation

In this lesson, we will install the Berkeley Internet Name Domain (BIND). This is the most widely used DNS server on the internet. Once you've installed BIND, you'll be able to set up new host entries in your own domain, which we'll use later when we explore web and email servers. The term "daemon" is just another way of saying "service," and is prevalent in the Linux/Unix world. You may recall that the name of the OpenSSH server is "sshd." The "d" on the end of the service name stands for "daemon." You'll also see this later with "httpd", a web server.

The Berkeley Internet Name Daemon

BIND is a DNS server developed by the *Internet Systems Consortium*, a non-profit organization that maintains several pieces of software critical to the operation of the internet. BIND has been around for many years, and has become the de-facto standard for name servers on the internet. Because of its extremely wide deployment in Linux and Unix server environments, we'll use BIND for this course.

Installing BIND

Before we go on, install BIND, using the **yum** package manager:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum install bind
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package bind.x86_64 32:9.7.3-8.P3.el6_2.2 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                               Arch      Repository                               Version
                               Size
=====
Installing:
  bind                                x86_64    32:9.7.3-8.P3.e
  16_2.2                                updates    3.9 M

Transaction Summary
=====
Install      1 Package(s)

Total download size: 3.9 M
Installed size: 7.0 M
Is this ok [y/N]: y
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : 32:bind-9.7.3-8.P3.el6_2.2.x86_64
                                                    1/1

Installed:
  bind.x86_64 32:9.7.3-8.P3.el6_2.2

Complete!
```

Yum may install more packages for you than just bind in order to resolve dependencies. Once you have installed BIND, we can get to configuration.

Initial Configuration

As it's currently installed, BIND is configured as a *caching only nameserver*, which means that it is only capable of querying other nameservers to fulfill your DNS queries. While this is a perfectly acceptable mode of operation in some circumstances, you actually want to be able to create and serve your own DNS entries. This means that you will need to reconfigure BIND as an *authoritative nameserver*. In order to make this change, you'll edit the **/etc/named.conf** file. Wait...what? Why named.conf and not bind.conf? Well, the actual daemon you will be running is called "named" and not "bind". The binary "named" is just a part of the BIND suite. So with that in mind, let's take a look at **/etc/named.conf**. Due to the permissions on this file, you'll need to use root privileges to view and edit it:

OBSERVE:

```
//
// named.conf
//
// Provided by Red Hat bind package to configure the ISC BIND named(8) DNS
// server as a caching only nameserver (as a localhost DNS resolver only).
//
// See /usr/share/doc/bind*/sample/ for example named configuration files.
//

options {
    listen-on port 53 { 127.0.0.1; };
    listen-on-v6 port 53 { ::1; };
    directory "/var/named";
    dump-file "/var/named/data/cache_dump.db";
    statistics-file "/var/named/data/named_stats.txt";
    memstatistics-file "/var/named/data/named_mem_stats.txt";
    allow-query { localhost; };
    recursion yes;

    dnssec-enable yes;
    dnssec-validation yes;
    dnssec-lookaside auto;

    /* Path to ISC DLV key */
    bindkeys-file "/etc/named.iscdlv.key";
};

logging {
    channel default_debug {
        file "data/named.run";
        severity dynamic;
    };
};

zone "." IN {
    type hint;
    file "named.ca";
};

include "/etc/named.rfc1912.zones";
```

This gives us a pretty good starting point. You only have to edit a couple of lines in the "options" stanza in order to make your server authoritative. First, look at the "listen-on" directive. This directive specifies both the port (the default being port 53) and the IP address(s) that named should listen on. We see here that it's configured to listen for incoming requests on localhost only. This doesn't make for a very useful authoritative DNS server because other hosts can't query the server for anything. In order to make your DNS server listen on all the configured interfaces in your machine, substitute "any" for "127.0.0.1".

CODE TO TYPE:

```
...
options {
    listen-on port 53 { any; };
    listen-on-v6 port 53 { ::1; };
...

```

Note

Named is extremely picky about line termination in its configuration file, `named.conf`. All lines within configuration stanzas should be terminated with a semicolon (;). Additionally, the closing curly bracket (}) at the end of a stanza should have a semicolon after it. Finally, any directive that includes a set of curly brackets, such as the "listen-on" directive, should have a semicolon after each specified value within the braces (for example, "listen-on port 53 { 172.16.10.1; 192.168.0.152; };").

Next, the "allow-query" directive informs `named` to which hosts' queries it may respond. That is, if a host does not match one of the addresses or networks listed in this directive, `named` will not respond to a query originating from that host. Like the default for the "listen-on" directive, this is set to respond only to queries from localhost. Again, this isn't exactly useful for an authoritative nameserver. Here you can use the value "any" again to allow all hosts to query your server. If you were so inclined, you could specify a network address, such as 172.16.0.0/12, in order to limit the allowed hosts to your local network. For now, go ahead and edit the "allow-query" directive to allow any host:

CODE TO TYPE:

```
...
    memstatistics-file "/var/named/data/named_mem_stats.txt";
    allow-query { any; };
    recursion yes;
...

```

Now you have settings that allow other hosts on other networks to query your DNS server. However, your DNS server doesn't have any DNS records for your domain to serve yet. In order to add DNS records, you must first add *zones*. DNS zones are logical groupings of DNS entries that generally encompass domains. For example, the domain that you are now in charge of, "username.unix.useractive.com" (substituting *username* with your username, of course), will have all of its information stored in its own zone. If you had an "in-addr.arpa" reverse lookup domain, that would have its own zone as well. Near the bottom of the file, you will see that there's already one zone stanza defined. This is the root zone, which contains information about the root DNS servers. Because the root servers are at the top of the DNS hierarchy, there aren't any servers we can ask about them. Therefore, DNS servers must have this zone defined in order for them to properly handle DNS queries for which they are not authoritative. After this entry, you can insert your own zone definitions.

Before you get into setting up your first zone, let's discuss *delegation*, because you won't be able to control your own domain without it! Delegation describes the act of a domain owner giving control of a subdomain to another entity. For example, the O'Reilly School of Technology (OST) controls the "useractive.com" domain, and therefore automatically controls any subdomain within it (like "unix.useractive.com"). This means that OST would normally control the domain that corresponds to "username.unix.useractive.com". Instead of OST managing it though, we have delegated responsibility for your "username.unix.useractive.com" to you. Now that you understand how you became the manager of your own domain, we can move on and create your first zone.

Your zone's configuration stanza in `named.conf` will be similar to the root zone's stanza with a few key changes. You can copy and paste the root zone entry and use it as a template. The new entry goes just below the current root zone entry. When you're finished, it will look like this:

OBSERVE: your new zone stanza, before any edits

```
zone "." IN {
    type hint;
    file "named.ca";
};

zone "." IN {
    type hint;
    file "named.ca";
};
```

The first change you'll make is to the zone name. When `named` gets a query, it gets the domain information from the query and matches it against the zones that it's responsible for serving out. This means your zone name needs to match your domain name (`username.unix.useractive.com`).

Next, set the "type" and the "file." There are two main types of zones, "master" and "slave." Since your server is the sole authoritative DNS server for your domain, you can set the type to "master."

Finally, the file directive tells `named` which text file on the system contains the DNS entries for that zone. This file path is relative to the path defined in the "directory" directive in the "options" stanza above (`/var/named`). You can name your zone file anything you'd like, but a good standard to follow is "db.zone_name." Also, you'll want to organize your zone files within `/var/named`, so place your zone file in the `/pz` subdirectory (short for "primary zones"—more on this later). Your file directive will look like this:

`pz/db.username.unix.useractive.com`. Don't forget to include the "db." in the file specification!

CODE TO TYPE:

```
zone "." IN {
    type hint;
    file "named.ca";
};

zone "username.unix.useractive.com" IN {
    type master;
    file "pz/db.username.unix.useractive.com";
};
```

And we've defined our first zone! Go ahead and write out the **`named.conf`** file and exit your text editor. Of course, you probably want to know all about the *primary zone* now, don't you? I appreciate your curiosity! Primary zones are zones that are stored and maintained locally on the DNS server. That is, the zone files are stored and edited on the server that is serving the zone. *Secondary zones* don't have local zone files. The data for a secondary zone is read directly from a server on which the same domain is listed as a primary zone. If the server that holds the primary zone goes down, the server with the secondary zone for that domain can continue to respond to DNS queries, but updates cannot be made to the domain on the secondary server. This gives DNS high availability capabilities, which is important because a failure in DNS can cause many problems for machines that depend on it. Because you are running the sole DNS server for your domain (and not too much *depends* on it so far), we will create only primary zones.

That's it for the first lesson on BIND. At this point, `named` is configured and ready to run; now all you need are some zone files! We'll create and edit those files in the next lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

BIND: Zone Files

We've installed BIND and configured `named` for your own private domain. In this lesson, we'll cover creating and editing the zone files that **named** uses to store the information for your domain. By the end of this lesson, you'll have a fully configured forward lookup zone for your domain.

Your First Zone File

Before you can create your zone file, you need to create the `/pz` directory in `/var/named` to house your primary zone files. Create this directory, copy the `/var/named/named.empty` file into the newly created directory, and rename it **db.username.unix.useractive.com** (substitute your username for *username*). You will need root level privileges to do this. You can use "sudo -s" to get a root shell if you'd like. You will have to modify the permissions of the new directory and the new zone file in order for the named server to be able to read them. The ownership of the directory and file should be "root:named."

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo -s
[root@username-m0 username]$ cd /var/named
[root@username-m0 named]$ mkdir pz
[root@username-m0 named]$ cp named.empty pz/db.username.unix.useractive.com
[root@username-m0 named]$ chgrp -R named pz
```

This will give you a good framework for your first zone. Go ahead and open the file for editing. At this point there really isn't much to see:

OBSERVE:

```
$TTL 3H
@      IN SOA  @ rname.invalid. (
                                0      ; serial
                                1D      ; refresh
                                1H      ; retry
                                1W      ; expire
                                3H )    ; minimum

NS     @
A      127.0.0.1
AAAA   ::1
```

At the top, you have a `$TTL` record. This defines the default *time to live* for this particular zone. You may recall that the TTL parameter defines how long a caching DNS server will cache a particular DNS entry. When a result is cached, subsequent queries for that particular entry receive the cached data rather than an authoritative answer. This reduces load on the authoritative server. Setting the TTL correctly for your domain requires careful consideration. Your TTL determines how quickly updates to your zone will propagate across the internet. If you have a very low TTL, cached DNS entries for your domain on other DNS servers will expire quickly and they will request a new authoritative answer from you more frequently. This means that changes you make to your domain will spread more quickly. A high TTL means it could take a while for your changes to hit other DNS servers on the net. While it may seem like setting a low TTL value is a good plan, keep in mind that a low value means that other DNS servers will have to make many more queries of your authoritative server. This is fine if you have a domain in which you expect relatively few lookups, but if you have a particularly busy domain, this can lead to a lot of unnecessary lookups and generate a considerable load.

For this particular domain, it's unlikely that you'll get many DNS queries, and you'll probably be making many changes, so you can pick a very low TTL value, such as 30 seconds. Busier sites may choose a TTL value of 5 minutes (600 seconds) or higher. While the `$TTL` parameter may define the default TTL value for the majority of record types in your zone, you will probably want to select custom TTL values for other record types. We'll cover this in detail a little later in the lesson. For now, edit the value of `$TTL` and set it to 30. The default unit for all time-related parameters in named configuration files is seconds, so you won't need to specify a unit.

CODE TO TYPE:

```
$TTL 30
@      IN SOA  @ rname.invalid. (
```

The next line we come to in your zone file is the SOA or *start of authority* line. The SOA is actually a whole entry that extends from this line to the closing parenthesis (you'll find it on the line immediately above "NS @"). This entry defines some parameters for your zone, most of which control how secondary DNS servers in your domain will handle transferring data and how long that data is valid. The first line in the SOA record defines the nameserver as well as the email address of the person responsible for the zone. The line will look like this:

OBSERVE:

DOMAIN	IN SOA	NAMESERVER.	EMAIL ADDRESS.
--------	--------	-------------	----------------

Note

Did you notice the periods at the end of the nameserver and email addresses? This is our way of telling `named` that these addresses are *Fully Qualified Domain Names* or *FQDN*'s for short. This will become extremely important when constructing reverse lookup zones later, so commit this information to your memory bank!

It looks like you have everything you need to construct the beginning of your SOA record now, but there are still two important details left to take into account, both of which hinge on the "@" character. The "@" character takes on a special meaning in a zone file. It is interpreted by `named` as a location where the \$ORIGIN value should be substituted. Generally speaking, unless the \$ORIGIN variable has been defined specifically in your zone file, it uses the name you assigned your zone in `named.conf` (which is your domain itself, "username.unix.useractive.com") by default. You can also define an \$ORIGIN explicitly in your zone file for clarity if you choose. Go ahead and set your \$ORIGIN after the \$TTL, but before your SOA record begins. Remember the trailing period!

CODE TO TYPE:

```
$TTL 30
$ORIGIN username.unix.useractive.com.
@      IN SOA  @ rname.invalid. (
```

While this step was not absolutely necessary, it makes your zone file easier to read and understand. Now that you know what the \$ORIGIN is, and understand the function of the "@" character, you can simplify your SOA record a bit. The next detail I'd like to cover is a little more subtle, but you may have already figured it out. The standard format for an email address is user@domain. You've just seen that `named` interprets the "@" symbol as the \$ORIGIN, not as the classic "at". This is problematic, but there is a basic workaround. `Named` expects an email address in this field of the SOA record, so it is happy to parse an email address in a non-standard format: username.domain. Therefore, "somedude@gmail.com" becomes "somedude.gmail.com." With all of this in mind, you can finally begin to create your SOA entry. When we delegated your domain to you, we set your nameserver's hostname to "ns.username.unix.useractive.com." Set the email address as the user "hostmaster." We'll have some fun with this later when we discuss email.

CODE TO TYPE:

```
$TTL 30
$ORIGIN username.unix.useractive.com.
@      IN SOA  ns.username.unix.useractive.com. hostmaster.username.unix.useractive.co
m. (
```

That was a lot of information to consider for only three lines of a zone file! Understanding some of the subtleties of these parameters will help you to set reasonable values for your domain. Starting your domain off right from the beginning will save you a lot of headaches down the road. We will not be editing the parameters found between the parenthesis in the SOA record, because they deal primarily with the way zone transfers are handled between primary and secondary DNS servers. Since you are only setting up a primary DNS server for your domain, these can be ignored.

Next, we'll create the actual DNS records for your domain. The first couple of records you have in your zone file are required for your domain to function properly; the remaining entries are up to you. First, clear out the three records currently at the bottom of the file to make room for your records:

CODE TO TYPE:

```
$TTL 30
$ORIGIN username.unix.useractive.com.
@      IN SOA  ns.username.unix.useractive.com. hostmaster.username.unix.useractive.co
m. (
                                0      ; serial
                                1D     ; refresh
                                1H     ; retry
                                1W     ; expire
                                3H )   ; minimum

----- NS ----- @
----- A ----- 127.0.0.1
----- AAAA ----- ::1
```

Replace those records with NS and MX records. NS records are used to store information about the authoritative nameservers for a domain; MX records hold information about the domain's mail exchangers. MX records are not required for your domain to function, but NS records are. An NS record will take this form:

OBSERVE:

DOMAIN	IN	NS	NAMESERVER
--------	----	----	------------

As above, you can substitute the "@" symbol for the domain, and your nameserver is **ns.username.unix.useractive.com**. Your MX record record will be similar, but with a key difference:

OBSERVE:

DOMAIN	IN	MX	PREFERENCE	MAILSERVER
--------	----	----	------------	------------

Here you must also specify a preference value for your mail server. This number must be an integer between 0 and 65536—the lower the number, the higher the preference. Most people will use 10 for their primary mailserver. The preference value, while not all that important when you only have one mailserver, becomes very important when you have multiple mailservers. This number determines the order in which your mailservers are used to process mail for your domain. That is, if the first defined mailserver is not available, the mailserver with the next highest priority will be tried, and so on until the message either goes through or the list of mailservers is exhausted. At large sites, it's not uncommon to have two or more mailservers. In fact, Google has five! Go ahead and add an NS and an MX record to your zone file:

CODE TO TYPE: adding NS and MX records to your zone

```
$TTL 30
$ORIGIN username.unix.useractive.com.
@      IN SOA  ns.username.unix.useractive.com. hostmaster.username.unix.useractive.co
m. (
                                0      ; serial
                                1D     ; refresh
                                1H     ; retry
                                1W     ; expire
                                3H )   ; minimum

@      IN      NS      ns.username.unix.useractive.com.
@      IN      MX      10 mail.username.unix.useractive.com.
```

Notice that we use periods at the end of our hostnames in order to designate them as being FQDNs. We're really making some progress here! In fact, we're close to having a fully functional zone file! All that remains is to add A records for both the NS and MX records. An A record is the record type that maps a hostname to an IP address. This is necessary because computers communicate with each other using IP addresses, not hostnames. In fact, that's the whole reason DNS exists in the first place! Below is the format for an A record:

OBSERVE:

HOSTNAME	IN	A	IP ADDRESS
----------	----	---	------------

Here we will use the short version of the hostname, rather than an FQDN. You could have used a shortened version of the hostname for the NS and MX records above, but there's nothing wrong with using the entire hostname. When you specify a hostname without a period at the end, named appends the \$ORIGIN variable automatically in order to create a FQDN. This is why you added periods to the end of your FQDNs earlier. If you hadn't, named would end up taking the hostname "foo.username.unix.useractive.com" and turning it into "foo.username.unix.useractive.com.username.unix.useractive.com"! Of course, this is not what you want. Go ahead and add entries for the hosts "ns" and "mail," as well as an entry called "m0," which will also refer to your machine. Use your machine's IP address in place of *n.n.n.n* for the records:


CODE TO TYPE:

@	IN	NS	ns.username.unix.useractive.com.
@	IN	MX	10 mail.username.unix.useractive.com.
ns	IN	A	n.n.n.n
mail	IN	A	n.n.n.n
m0	IN	A	n.n.n.n

Now your zone is ready for action! You finally get to test to see if everything is working properly. Write out your zone file and use the service command to start named.

INTERACTIVE SESSION:

```
[username@username-m0 pz]$ sudo service named start
Starting named: [ OK ]
[username@username-m0 pz]$
```

If you get any errors from this command, contact your mentor for assistance. If you don't get any errors, great! In order to test your new DNS server out, start a new shell on cold by clicking the **New Terminal**  button. Once you have logged in, use dig to look up the NS and MX records for your domain.

INTERACTIVE SESSION:

```
cold1:~$ dig username.unix.useractive.com ANY

; <<>> DiG 9.7.3-P3-RedHat-9.7.3-8.P3.el6_2.2 <<>> username.unix.useractive.com ANY
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17827
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 2

;; QUESTION SECTION:
;username.unix.useractive.com. IN      ANY

;; ANSWER SECTION:
username.unix.useractive.com. 30 IN      SOA      ns.username.unix.useractive.com. hostmas
ter.username.unix.useractive.com. 0 86400 3600 604800 10800
username.unix.useractive.com. 30 IN      MX       10 mail.username.unix.useractive.com.
username.unix.useractive.com. 30 IN      NS       ns.username.unix.useractive.com.

;; AUTHORITY SECTION:
username.unix.useractive.com. 30 IN      NS       ns.username.unix.useractive.com.

;; ADDITIONAL SECTION:
mail.username.unix.useractive.com. 30 IN A       n.n.n.n
ns.username.unix.useractive.com. 30 IN A       n.n.n.n

;; Query time: 4 msec
;; SERVER: 10.0.131.16#53(10.0.131.16)
;; WHEN: Thu Mar 29 14:34:27 2012
;; MSG SIZE rcvd: 178
```

If you don't see something similar to what we've got in our example, contact your mentor for help. When you've confirmed that your DNS server is working properly, return to your machine and open up the zone file for editing again. This time add one more records: another a *CNAME* record. A CNAME record, short for *canonical name*, acts as an alias for another entry. For example, say you have a host "foo," but you also want the name "bar" to resolve to the same address as "foo." You can create a CNAME record that points "bar" at "foo," so that a lookup for "bar" will return the information for "foo." A CNAME record uses this format:

OBSERVE:

ALIAS	IN	CNAME	HOSTNAME
-------	----	-------	----------

At the bottom of your zone file, add a CNAME record that points "www0" to your machine, "m0".

CODE TO TYPE:

ns	IN	A	n.n.n.n
mail	IN	A	n.n.n.n
m0	IN	A	n.n.n.n
www0	IN	CNAME	m0

And write the file out. In a real setting, you would likely have different addresses for all of these A records, but because you only have one machine at the moment, one address will have to do. In order for these settings to take effect, restart **named** using the service command, and then use **nslookup** to verify that your new records have been added:

INTERACTIVE SESSION:

```
[username@username-m0 pz]$ service named restart
Stopping named: .[ OK ]
Starting named: [ OK ]
[username@username-m0 sites]$ nslookup www0.username.unix.useractive.com localhost
Server:          localhost
Address:         127.0.0.1#53

www0.username.unix.useractive.com      canonical name = m0.username.unix.useractive.com
.
Name:   m0.username.unix.useractive.com
Address: n.n.n.n
```

That's it for your first DNS zone file! In the next lesson we'll cover building Apache's httpd webserver software from source code. See you then!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Building Software

This lesson will cover the process of building a piece of software, in this case the Apache web server, from source. From locating and downloading the source code to installing it after it's been compiled, we'll cover it all.

Locating and Downloading the Source

Every piece of software that you use on a daily basis was at one time just source code. This is what software developers write before *compiling* it into a binary. While a lot of software projects don't make their source code available to end users, many projects do. A project with source code that is available to view and edit is called *open source*. While you may not intend to modify source code, building software from source gives you the option to include or exclude functionality, and more control over the version of software you use. The ability to build software from source is an important part of a system administrator's toolkit.

Let's get straight to work and retrieve the source code we want to build. We'll be using the Apache webserver as our example in this course. There are many different locations (called *mirrors*) from which you can download the source, but we allow you to download software from one mirror only, `ftp.osuosl.org`. For a full list of Apache mirrors, see the [list on apache.org](http://www.apache.org). You may recall from the last course that we strongly discourage the use of FTP in most cases. However, we're going to make an exception to that advice in this instance. In order to mitigate your risk for sending sensitive data out over the internet unencrypted, you will use *anonymous FTP*; that is, you will log into the remote FTP server without giving a password. (Many FTP servers that serve as source code mirrors allow this kind of connection.) Make sure you are in your home directory on your Linux learning environment machine and start an FTP session with `ftp.osuosl.org`. When prompted for a username, give the name **anonymous**. Press **Enter** when prompted for a password. Once you're logged in, change directory to `/pub/apache/httpd`. You can list the files in this directory by running `ls`. Use the `ftp` command **get** to retrieve the latest version of `httpd` from the 2.2 branch (in the below example, 2.2.26). The filename will have a `.tar.bz2` extension.

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ftp ftp.osuosl.org
Trying 64.50.233.100...
Connected to ftp.osuosl.org (64.50.233.100).
220-----
220-                      F T P . O S U O S L . O R G
220-                      Oregon State University
220-                      Open Source Lab
220-
220-    Unauthorized use is prohibited - violators will be prosecuted
220-----
220-
220-          For more information on our services visit:
220-          http://osuosl.org/hosting/
220-
220-          We offer these particular files simply because users
220-          have asked for them.  If you would like another archive offered
220-          here on this ftp mirror, please ask support@osuosl.org
220-
220-    Questions/Comments/Suggestions/Concerns - support@osuosl.org
220-----
220
Name (ftp.osuosl.org:username): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/apache/httpd
250 Directory successfully changed.
ftp> get httpd-2.2.27.tar.bz2
local: httpd-2.2.27.tar.bz2 remote: httpd-2.2.27.tar.bz2
227 Entering Passive Mode (64,50,236,52,176,109).
150 Opening BINARY mode data connection for httpd-2.2.27.tar.bz2 (5378934 bytes).
226 Transfer complete.
5378934 bytes received in 1.34 secs (4010.07 Kbytes/sec)
ftp> quit
221 Goodbye.
```

Now you have the source code archive for Apache's httpd server! Before you can do anything with it though, you have to *extract* the files from the archive. The file extension indicates which kind of archive the file is; in this case it's a *bzip2 compressed tarball*. The word "tarball" is derived from the tool used to create the archive, **tar**, which is short for **t**ape **a**rchive. Tarballs are common in the Linux world, though in most cases they will be compressed. Generally, tarballs are compressed with either the *gzip* algorithm (giving the extension *.tgz* or *.tar.gz*) or in the case of the archive we're working with, *bzip2*. While it's possible to decompress the archive first and then extract the files from it, it's convenient to do both operations at once. The *tar* utility allows us to do that by specifying flags that instruct it to run the archive through the chosen decompression algorithm first. Here are some of the most commonly used tar flags:

Flag	What It Does
x	Extract the specified archive.
j	Run the archive through the bzip2 decompression algorithm before extracting.
z	Run the archive through the gzip decompression algorithm before extracting.
v	Operate verbosely. That is, print out the file names as they are extracted.
f	Specify the filename of the archive to operate on. This flag should always come last.

Note Tar is another utility that accepts flags with or without a leading hyphen.

We want to extract our archive after decompressing it with the *bzip2* algorithm, we want to specify the filename of the archive and, just for fun, we'll make the output verbose. That means the command we want will look like this: **tar xvjf**

archive.tar.bz2. You can substitute the filename of the archive you downloaded for *archive.tar.bz2*. Go ahead and unarchive the source using `tar`. (You'll see lots of output because you included the `v` flag.)

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ tar xvjf httpd-2.2.27.tar.bz2
...
httpd-2.2.27/VERSIONING
httpd-2.2.27/README
httpd-2.2.27/LAYOUT
httpd-2.2.27/buildconf
httpd-2.2.27/.gdbinit
[username@username-m0 ~]$
```

Now that we've got the source for Apache's `httpd` unarchived, we can move on to building it!

Building Software From Source

Generally, the various processes for building software from source are similar to one another regardless of which software you're building on your machine. The build process has been more or less standardized over the years, and while some projects will execute tasks in a unique manner, most will use the **make** utility to handle the heavy lifting. In order to accommodate a large number of inconsistent platforms, software usually ships with a script called **configure**. This script can take some arguments to customize the build process, and then generate something called a *makefile* based on information it determines about your system. The *makefile* is then used as a kind of playbook by `make` to build and install the software. Luckily, Apache's `httpd` software ships with fairly sensible defaults for its build, so we don't have to specify non-default values to `configure`. If you haven't done so already, change directory into the unarchived source directory. Then, run the `configure` script. It's not in your path, so you'll have to run it as `./configure` to specify that you want to run the `configure` script in the present working directory. (You'll get a significant amount of output, and the script may take a while to complete.)

INTERACTIVE SESSION:

```
[username@username-m0]$ cd httpd-2.2.27
[username@username-m0 httpd-2.2.27]$ ./configure
checking for chosen layout... Apache
checking for working mkdir -p... yes
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking target system type... x86_64-unknown-linux-gnu
...
config.status: creating build/rules.mk
config.status: creating build/pkg/pkginfo
config.status: creating build/config_vars.sh
config.status: creating include/ap_config_auto.h
config.status: executing default commands
[username@username-m0 httpd-2.2.27]$
```

If you don't see any errors, a *makefile* has been generated successfully by `configure` and you can start to build the software! Fortunately, the `make` command makes the process fairly straightforward. Just run `make` and `httpd` starts building for you. Again, you will get quite a bit of output and it will probably take far longer than `configure` to complete. (Now might be a good time to get up and stretch your legs!)

INTERACTIVE SESSION:

```
[username@username-m0 httpd-2.2.27]$ make
Making all in srclib
make[1]: Entering directory `/home/dbasset1/httpd-2.2.27/srclib'
Making all in pcre
make[2]: Entering directory `/home/dbasset1/httpd-2.2.27/srclib/pcre'
make[3]: Entering directory `/home/dbasset1/httpd-2.2.27/srclib/pcre'
...
gcc -pthread -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -I/home/dbasset1/httpd-2.2.27/
srclib/pcre -I. -I/home/dbasset1/httpd-2.2.27/os/unix -I/home/dbasset1/httpd-2.2.27/s
erver/mpm/prefork -I/home/dbasset1/httpd-2.2.27/modules/http -I/home/dbasset1/httpd-2
.2.27/modules/filters -I/home/dbasset1/httpd-2.2.27/modules/proxy -I/home/dbasset1/ht
tpd-2.2.27/include -I/home/dbasset1/httpd-2.2.27/modules/generators -I/home/dbasset1/
/modules/mappers -I/home/dbasset1/modules/database -I/usr/include/apr-1 -I/home/dbass
ett1/modules/proxy/./generators -I/home/dbasset1/modules/ssl -I/home/dbasset1/mod
ules/dav/main -c /home/dbasset1/server/buildmark.c
/usr/lib64/apr-1/build/libtool --silent --mode=link gcc -pthread -o httpd modul
es.lo buildmark.o -export-dynamic server/libmain.la modules/aaa/libmod_authn_file.la mo
dules/aaa/libmod_authn_default.la modules/aaa/libmod_authz_host.la modules/aaa/libmod_a
uthz_groupfile.la modules/aaa/libmod_authz_user.la modules/aaa/libmod_authz_default.la
modules/aaa/libmod_auth_basic.la modules/filters/libmod_include.la modules/filters/libm
od_filter.la modules/loggers/libmod_log_config.la modules/metadata/libmod_env.la module
s/metadata/libmod_setenvif.la modules/metadata/libmod_version.la modules/http/libmod_ht
tp.la modules/http/libmod_mime.la modules/generators/libmod_status.la modules/generator
s/libmod_autoindex.la modules/generators/libmod_asis.la modules/generators/libmod_cgi.l
a modules/mappers/libmod_negotiation.la modules/mappers/libmod_dir.la modules/mappers/l
ibmod_actions.la modules/mappers/libmod_userdir.la modules/mappers/libmod_alias.la modu
les/mappers/libmod_so.la server/mpm/prefork/libprefork.la os/unix/libos.la -lm /home/db
asset1/srclib/pcre/libpcre.la /usr/lib64/libaprutil-1.la -ldb-4.7 -lexpat -ldb-4.7 /u
sr/lib64/libapr-1.la -lpthread
make[1]: Leaving directory `/home/dbasset1/'
[username@username-m0 ]$
```

If you get any errors, contact your mentor for assistance; if possible, include the specific errors from the output of the make command. If you don't get any errors, then you can move on to installation! Again, make will handle all of the work for us. However, because you'll be writing files to a system directory and not your home directory, you'll have to use sudo. Run **make** with the **install** subcommand to copy the necessary files to your system:

INTERACTIVE SESSION:

```
[username@username-m0 ]$ sudo make install
...
Installing man pages and online manual
mkdir /usr/local/apache2/man
mkdir /usr/local/apache2/man/man1
mkdir /usr/local/apache2/man/man8
mkdir /usr/local/apache2/manual
make[1]: Leaving directory `/home/dbasset1/'
[username@username-m0 ]$
```

The entire Apache httpd software suite has been installed to `/usr/local/apache2`. It's good practice to install software that you compile into a subdirectory within `/usr/local`. This keeps your software separate from software that's installed using your system's package manager, making it easier to maintain or remove without disturbing other software. Before we can call this installation a success, we need to test it. In order to do that, we'll need to start the httpd server. Although some prepackaged versions of httpd come with an init script, the version that you built does not, so you'll use the **apachectl** utility to control httpd. **Apachectl** is found in `/usr/local/apache2/bin`, which, as you may have guessed, is the directory inside the httpd install that holds httpd and its associated binaries.

Using **apachectl** is similar to using the **service** command. It takes subcommands such as **start** and **stop**, which do exactly what you would expect. Let's start your httpd server for the first time by running **apachectl start**!

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo /usr/local/apache2/bin/apachectl start
```

If your server is running right, you will be able to go to **`http://username-m0.unix.useractive.com`** in your web browser (substituting your username, of course) and you'll be greeted with a short message that will let you know how you did. If you see it, then congratulations! You've just compiled and installed your first webserver!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

HTTPD: Introduction and Initial Configuration

This lesson covers some httpd server concepts and some initial configuration changes. By the end of this lesson you will have your first name-based virtual host working!

An Introduction to Web Servers

Web servers use the *HTTP* or *HyperText Transport Protocol* to deliver web pages to your browser. This protocol allows client software (such as Firefox or Internet Explorer) to send a request to a web server, which in turn processes the request and returns data to the client. This is true not only for simple HTML documents, but also for dynamic content such as web sites written with PHP or CGI (we'll discuss dynamic content later). HTTP outlines several *request methods* that clients can use to retrieve or upload data. Based on user input, the client constructs and sends an appropriate message, including the necessary request method, along with any other ancillary data that the server needs to fulfill the request. The most common method is the *GET* method. As the name implies, this method is used to retrieve data, such as a web page, from the web server. In the simplest static HTML web page, a GET request will have the browser return the unrendered HTML to the client, along with the appropriate *status code*. The client uses the status code sent by the server to determine what should be done with the data it receives. If the status code indicates that the request was fulfilled without any issues, the client then renders the HTML it receives from the web server into the intended web page.

Another common request method, *POST*, allows clients to send form data to the web server for processing. The processing can include, but is not limited to, generating an email or providing data for variables in a dynamic web site written in PHP, CGI, or another language. There are other less commonly used methods available, but we'll skip them for now. In addition to providing methods for sending and retrieving data, HTTP outlines several fields for additional information that can be supplied to the server. These fields appear in the *header* of the HTTP request. One important field that's required as of HTTP version 1.1, is called *Host*. This field allows one web server to host multiple websites on the same TCP port by providing a way for the client to specify which site it wants to access. We will use this feature later when we set up *name-based virtual hosts* on your machine.

Now that we've been introduced to the protocol used to communicate with web servers, we can address what a web server does when it receives a request. When a GET message is received, it contains a filename (and if it's HTTP/1.1 compliant, a hostname). A basic HTTP GET request for your web server might look like this:

OBSERVE:

```
GET /index.html HTTP/1.1
Host: username-m0.unix.useractive.com
```

The web server interprets this as a request for the file **index.html** for the virtual host **username-m0.unix.useractive.com**. The web server then reads the **index.html** file from its filesystem and sends it back to the client, along with a status code and some other details. In the GET request, the filename is given as "/index.html", but this does not mean that the file is located in the root directory of the filesystem. Rather, all of the requested files' locations are relative to a specific directory that is defined in the web server's configuration. This directory is called the *document root*. Each virtual host running on a web server will have its own document root in order to keep everything organized. The response to the above request would look something like this:

OBSERVE:

```
HTTP/1.1 200 OK
Date: Mon, 02 Apr 2012 20:19:12 GMT
Server: Apache/2.2.22 (Unix)
Last-Modified: Sat, 20 Nov 2004 20:16:24 GMT
ETag: "e7c9-2c-3e9564c23b600"
Accept-Ranges: bytes
Content-Length: 44
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

Here you see your first HTTP status code on the first line. A response of 200 means that the request was handled successfully. You may also be familiar with another error code from normal web browsing, 404. This code denotes that the requested item could not be located on the server. Other lines of interest are "Content-Length" and "Content-Type." These lines help the browser deal with the data it receives in the message by telling it the kind of content it is and how long it should be. With the above information in mind, you can configure your first virtual hosts on your web

server!

Initial Configuration Changes

Each virtual host that you configure will need its own document root. Currently, your httpd is configured to use **/usr/local/apache2/htdocs**, which is located inside the httpd installation directory. This is not really what you want. While you can choose to locate your document root anywhere on your system, most installations will use a subdirectory located in **/var/www**. In fact, many prepackaged versions of Apache's httpd come configured to use this directory, but since you compiled your own httpd, this directory has not yet been created. Go ahead and create it now:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo mkdir /var/www
```

Now create another directory inside **/var/www** to use as document root for the virtual host you are going to set up. In the BIND server lesson, you set up a CNAME called **www0** that pointed to your machine. This will be your first virtual host. While there is no standard for naming virtual host document roots, it's a good idea to name them based on the hostnames of the virtual hosts themselves in order to stay organized. Go ahead and create a directory within **/var/www** named **www0** and set its owner to yourself:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo mkdir /var/www/www0
[username@username-m0 ~]$ sudo chown username:username /var/www/www0
```

Excellent! Now you can create a test web page to host in this location. HTML skills are not required here. We can just create a text file named **index.html** in the appropriate directory. In case you were wondering, **index.html** is the default file that's loaded by the web server if no filename is specified by the client in a URL. That is, if a GET request asks for a directory rather than a file name, the web server looks for an **index.html** file in the specified directory. Change directory into **/var/www/www0** and create an **index.html** file. A quick way to do this is using **echo**, but use single quotation marks to surround your text. If you use double quotation marks, the shell will interpret your exclamation points as an instruction to run an event from its history!

INTERACTIVE SESSION:

```
[username@username-m0 www0]$ echo 'Hello world! This is www0 speaking!' > index.html
```

Once you have your file in place, you can configure httpd to look it up. Your initial configuration changes will happen in **/usr/local/apache2/conf/httpd.conf**. First, you must disable the current website that httpd displays. Open **httpd.conf** to edit (you'll need root privileges) and locate the line that begins with **DocumentRoot**. You will no longer use this document root, and in fact you will configure separate document roots for each of your virtual hosts, so you can *comment it out*. For this particular configuration file, a line that begins with a hash (#) is interpreted as a comment, so you can just place a "#" in front of the **DocumentRoot** line:

Commenting Out the DocumentRoot:

```
...
#
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#
#DocumentRoot "/usr/local/apache2/htdocs"
...
```

Keep the file open. Next, enable name-based virtual hosting by telling httpd which TCP socket to listen on for name-based virtual host requests. Navigate to the bottom of **httpd.conf** and insert the following on a new line:

CODE TO TYPE:

```
...
<IfModule ssl_module>
SSLRandomSeed startup builtin
SSLRandomSeed connect builtin
</IfModule>

NameVirtualHost *:80
```

Don't close the file yet! This statement tells httpd to expect requests on port 80 for all configured IP addresses to be handled as name-based virtual host requests. That is, when an HTTP GET request is received on port 80 for any of the interfaces on your machine, httpd will use the "Host" field in the GET request to determine which host (and consequently which document root) to use. Now you could define each virtual host in the "httpd.conf" file, but there is a much better way to handle individual web site configurations. Apache's httpd allows external configuration files to be included in the main configuration using *include statements*. This means that you can create a separate configuration file for each site you plan to host, allowing for much cleaner organization of your configurations. Add this include statement to the end of httpd.conf:

CODE TO TYPE:

```
...
<IfModule ssl_module>
SSLRandomSeed startup builtin
SSLRandomSeed connect builtin
</IfModule>

NameVirtualHost *:80

Include conf/sites/*.conf
```

This statement instructs httpd to load any configuration files it finds in the **conf/sites** directory. But what's the deal with that path? It's a relative path, but what is it relative to? In order to find out, search for the **ServerRoot** variable in the httpd.conf file. You'll see that it's set to **/usr/local/apache2** in your installation. Any relative path that's defined anywhere in httpd.conf is considered to be relative to this directory, therefore your site configuration files must be located in **/usr/local/apache2/conf/sites**. However, this directory does not currently exist, so you'll need to create it. Save and close **httpd.conf**, then issue the following commands:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo mkdir /usr/local/apache2/conf/sites
[username@username-m0 ~]$ cd /usr/local/apache2/conf/sites
[username@username-m0 sites]$
```

You're almost there! You just need to create the configuration file for your virtual host. This configuration file can have any name you choose, but for clarity's sake, let's name it for the site it defines, in this case **www0**. Open a new file named **www0.conf** to be edited with your favorite text editor, with root privileges. This file will contain the configuration for your site located at **www0.username.unix.useractive.com**. Enter this text:

CODE TO TYPE:

```
<VirtualHost *:80>
    ServerName www0.username.unix.useractive.com
    ServerAlias www0
    DocumentRoot /var/www/www0

    <Directory "/var/www/www0">
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Let's look at the syntax of this configuration file. Configuration stanzas are formed by opening and closing tags, similar

to the style of HTML. Individual options that apply within the scope of the stanza appear on their own lines, and stanzas can be nested within other stanzas. The main stanza in your file defines a virtual host instance. The VirtualHost directive must be accompanied by a list of TCP sockets on which to listen, in this case port 80 on all addresses. The first two directives within the virtual host stanza define the host names for which this particular virtual host will respond. These values are compared against the contents of the "Host" variable in an incoming HTTP GET request to determine which virtual host matches. The ServerName directive should be set to the virtual host's FQDN, whereas the ServerAlias can be set to any other name (FQDN or short host name) to which the virtual host may respond. Next, the DocumentRoot directive specifies the document root for this particular virtual host. This is set to point to the directory you created (and put an index.html file into) earlier. Finally, the Directory configuration stanza is used to set directory-wide options. In this example, we are using it to override the default restrictive policy set by httpd's main configuration file. (We'll discuss access control in much more depth in a later lesson.)

Once you're confident that you understand this configuration file, write the file out and quit the editor. Then restart your web server using the apachectl utility. In addition to the "start", "stop," and "restart" actions, there is a "graceful" action you can use, which restarts the httpd processes on your machine without terminating any existing HTTP sessions. Old sessions will continue to use the previous configuration and new sessions will use the new configuration. This is generally the preferred method for restarting httpd as it eliminates downtime. Go ahead and do a graceful restart of your web server now using apachectl:

INTERACTIVE SESSION:

```
[username@username-m0 sites]$ sudo /usr/local/apache2/bin/apachectl graceful
```

You can test your first virtual host by visiting **www0.username.unix.useractive.com**. If you are greeted with the message "Hello world! This is www0 speaking!", then you have set everything up correctly! In the next lesson, we'll cover some more features of httpd.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

HTTPD: Configuration

In this lesson on Apache's httpd, we'll go over ways to allow users to create webpages in their home directories and set up access controls. These capabilities are important in multi-user environments where people want personal webpages that are password protected.

Enabling Userdir in Httpd

We'll start by enabling userdir websites, and then work on securing these sites. First, create another virtual host, this time corresponding to **m0.username.unix.useractive.com**. You already have a DNS entry for this host, so you won't need to add anything to your zone file. Open a virtual host configuration file for editing in **/usr/local/apache2/conf/sites** and name it **userdir.conf**. Add this configuration:

CODE TO TYPE:

```
<VirtualHost *:80>
    ServerName m0.username.unix.useractive.com
    ServerAlias m0
    UserDir public_html

    <Directory "/home/*/public_html">
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Most of this is familiar to you, except for the UserDir directive. It tells httpd that it should treat this virtual host as one that allows userdir lookups, and to use the directory **public_html** inside users' home directories as the location of user web pages. Userdir websites can then be accessed by visiting URLs that follow the pattern **http://m0.username.unix.useractive.com/~some_user**, where **some_user** is the username of the person whose website you want to see. When you have this configuration written, go to your home directory and create a **public_html** directory and then create an **index.html** file containing the sentence, **This is a message from my userdir website!** as shown:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ chmod 711 /home/username
[username@username-m0 ~]$ mkdir public_html
[username@username-m0 ~]$ cd public_html/
[username@username-m0 public_html]$ echo 'This is a message from my userdir website!' >
index.html
```

First, we want to change the permissions of the home directory to allow Apache access to the public_html directory. Now, restart httpd gracefully and navigate to **http://m0.username.unix.useractive.com/~username** in your browser. If you don't see the message you just created, contact your mentor and go over any errors that you received. If you do see the message, then you have configured httpd successfully to provide userdir service! In the next section you'll learn techniques to secure websites using features built into Apache's httpd.

Per-Directory Access Controls

Httpd can define access control policies on a per-directory basis. This is especially useful for securing specific directories on a website by limiting access. We've already seen directives used within the directory configuration stanza to relax some of the default restrictive policies set by the main httpd configuration. However, we didn't explain what those directives did.

OBSERVE:

```
<Directory "/var/www/www0">
    Order allow,deny
    Allow from all
</Directory>
```

The "Order" directive specifies the order in which `allow` and `deny` rules are processed. Here, the order is set to process allow rules first, then deny rules. For `httpd`, the last rule that matches is the rule that is applied to a particular request. For example, suppose you want to visit your website at `www0.username.unix.useractive.com` from a machine with the IP address 199.27.144.74. This configuration would allow a successful HTTP connection to occur:

OBSERVE:

```
<Directory "/var/www/www0">
    Order deny,allow
    Allow from 199.27.144.74
    Deny from all
</Directory>
```

However, this configuration would cause `httpd` to return a permission denied error:

OBSERVE:

```
<Directory "/var/www/www0">
    Order allow,deny
    Allow from 199.27.144.74
    Deny from all
</Directory>
```

It all comes down to the order in which the rules are processed. In the successful example, the deny rules are processed first, meaning that the "Deny from all" is encountered before the "Allow from 199.27.144.74." Because the "Allow from 199.27.144.74" is the last rule to match, it's the rule that is applied. In the unsuccessful example, the order is reversed, so the "Deny from all" is the last rule that matches, and so it's the rule that is applied to the connection. Note that the order of the rules in the file has nothing to do with the order in which they are processed. Only the Order directive can change the rule processing order.

Fortunately, the burden of managing directory access does not fall entirely on the administrator. Apache's `httpd` (as well as several others) provide a mechanism by which users can define their own per-directory options, including access controls. When this feature is enabled, directives regarding per-directory configuration can be placed in a file that resides in the directory to be configured. That is, if you wanted to specify a non-default configuration for the directory `/var/www/www0/files`, you could place a file called `.htaccess` containing configuration directives into `/var/www/www0/files`. `Httpd` parses the pertinent `htaccess` files when HTTP requests for content in that particular directory are received. This doesn't really tell the full story, though.

Let's consider an example URL, `http://your.website.com/files/documents/top_secret/`, with a document root located at `/var/www/your.website.com/`. If you were to make a request for a document at this location, and `htaccess` was enabled on the webserver, `httpd` would first look for an `htaccess` file at `/var/www/your.website.com/`, then `/var/www/your.website.com/files/`, and so on. Because of that process, per-directory options are *inherited* by subdirectories. So, if you put an `htaccess` file in `/var/www/your.website.com/files/documents/top_secret` in order to limit access to the directory, any subdirectories within that directory would be subject to the same options. To get a look at `htaccess` in action, let's enable its use for your `userdir` virtual host. We'll do that with the "AllowOverride" directive. Open the "userdir.conf" site configuration for editing and make these changes:

CODE TO TYPE:

```
<VirtualHost *:80>
    ServerName m0.username.unix.useractive.com
    ServerAlias m0
    UserDir public_html

    <Directory "/home/*/public_html">
        Order allow,deny
        Allow from all
        AllowOverride limit
    </Directory>
</VirtualHost>
```

The "AllowOverride" directive specifies which options can be used in an htaccess file in the given directory. In this case, the **limit** set of options includes the **Order** directive, along with the **Allow** and **Deny** directives. Write the file out and gracefully restart httpd. Then, go into the /public_html directory in your home directory. Before you make any changes, navigate to **http://m0.username.unix.useractive.com/~username** in the browser and note the results. You should get a "permission denied" error. This happens because you have removed the rules that previously allowed you to access the site, and now httpd is defaulting to its more restrictive policy. Now open the file **.htaccess** in your public_html directory for editing and add this to it:

CODE TO TYPE:

```
Order deny,allow
Allow from all
```

Reload **http://m0.username.unix.useractive.com/~username**. If you receive any errors, contact your mentor. If not, you are using htaccess successfully to control access to your public_html directory. While this example may not be wildly exciting, there are more practical applications of htaccess files, such as one that allows users to password-protect directories in their websites. We'll discuss this shortly, but first we have to cover a security issue with ".htaccess" files.

If someone on the web was able to download the contents of your **.htaccess** file, it could allow them to circumvent any measures you have put in place to restrict access to your directory. For this reason, the main server configuration (httpd.conf) includes a statement that, by default, disallows files that have names that start with ".ht" from being downloaded. This may not always be the case if you're using a precompiled version of httpd, so always check to see if a rule like this exists:

OBSERVE:

```
<FilesMatch "^\.ht">
    Order allow,deny
    Deny from all
    Satisfy All
</FilesMatch>
```

The above stanza is a "FilesMatch" configuration, which takes a regular expression as a criteria to match files against before they are given the OK to be downloaded by a client. You may recall from the brief regular expressions introduction in Systems Administration I that a **caret (^)** denotes that the pattern should match only against the beginning of a string. Here, we want to find any string that starts with **.ht**. The **** before the period in ".ht" is necessary to make httpd parse the period as a period and not a wildcard character. The directives inside the stanza have the same meaning as the ones used above to provide directory-level access control. The **Satisfy** directive, generally used to determine how multiple restrictions should be applied to a resource (for example, you have configured both host-based ACLs and password authentication), ensures that regardless of any other ACLs set elsewhere, the **Deny from all** will be obeyed.

Now let's move on to configuring password authentication. In order to do this, you must add to the list of items in the **AllowOverride** directive in your virtual host configuration. Re-open your configuration for your userdir virtual host and add this:

CODE TO TYPE:

```
<VirtualHost *:80>
    ServerName m0.username.unix.useractive.com
    ServerAlias m0
    UserDir public_html

    <Directory "/home/*/public_html">
        AllowOverride limit authconfig
    </Directory>
</VirtualHost>
```

When that's finished, you'll need to restart httpd gracefully for the changes to take effect. Next, we go back to editing the ".htaccess" file in your home directory in order to add the directives necessary to enable password protection:

CODE TO TYPE:

```
Order deny,allow
Allow from all

AuthType Basic
AuthName "Top Secret Area"
AuthUserFile /home/username/public_html/.htpasswd
Require valid-user
```

Note

Another advantage of htaccess files is that when you change their contents, httpd does not need to be restarted in order for the changes to take effect.

Let's break down these statements. The first added directive **AuthType** is set to **Basic**. This tells httpd which type of authentication to use. The **Basic** authentication type is supported under the largest number of browsers, so it is the recommended method. Next, the **AuthName** directive gives a title to the area that is being protected by authentication. This name is used in the pop-up authentication box in your browser, so it should describe the contents of this location (for example, "Top Secret Documents" or "MyBigCo's Sensitive Data"). The **AuthUserFile** directive defines a file that contains username-to-password mappings to authenticate against. If the path given is not absolute, it is considered to be relative to the ServerRoot (defined in httpd.conf), so you'll likely want to make this an absolute path. Choosing a name that starts with ".ht" is good practice, because if you have a **FilesMatch** rule that excludes .htaccess files from being downloaded, it will also prevent your password file from being downloaded. Finally, the **Require** directive informs httpd which set of users should be allowed access. Here, a **valid-user** is any user that appears in the defined file and authenticates successfully. You can also use the **Require** directive to limit access to specific users or groups.

When you finish editing your **.htaccess** file, create the password file and populate it with users. For this task, Apache provides the **htpasswd** command. The format of this command is **htpasswd flags filename username**. For now, the only flag you need to worry about is **-c**, which is used to create the named password file. If you are adding a user to an existing password file, this flag can be omitted. Create a password file with an entry for yourself. Do not use your systemwide password.

INTERACTIVE SESSION:

```
[username@username-m0 public_html]$ /usr/local/apache2/bin/htpasswd -c .htpasswd username
New password:
Re-type new password:
Adding password for user username
```

Now visit **http://m0.username.unix.useractive.com/~username** again. You may have to refresh your browser in order to get the password prompt to appear. When you authenticate successfully, you're done! Congratulations on setting up your first password-protected web site!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Postfix: Introduction

This lesson introduces some of the concepts behind *SMTP*, the *Simple Mail Transfer Protocol*. Once you have an understanding of some of the basic concepts of internet mail, we'll introduce a very popular mail transfer agent, Postfix.

An Introduction to Mail Servers

In previous lessons regarding DNS, we introduced the term *mail exchanger*. Mail exchangers are servers whose job is to relay and deliver email. Mail exchangers on the Internet use SMTP to route and deliver email to the appropriate destination. There are, however, other protocols used in the process of mail delivery. Client email software will generally use SMTP to send outgoing messages, but most software will use either *POP* (the Post Office Protocol) or *IMAP* (the Internet Message Access Protocol) to access email messages stored on a mail server. There are key differences in these protocols that we will discuss later. For now, we will concentrate on SMTP.

SMTP, like HTTP, just describes the method that's used to transmit messages and not the content of the messages. At its core, SMTP is a very simple protocol, though there have been a few extensions added to the protocol to enable features like restrictions on relay access (more on this later). An SMTP transaction consists of a series of commands sent to the server and responses from the server, beginning with the command **HELO**, which is used to identify the host that the message is originating from. Once the sending host has been identified, the mail address of the sending user is reported to the server using the **MAIL** command. This data will be used by the mail exchanger to notify the original sender of the message if their message is undeliverable. Next, the **DATA** command is issued, followed by the headers and body of the email message itself. Finally, when the data stream is terminated by a specific sequence of characters, the SMTP connection can be closed by issuing the **QUIT** command. Below is a sample SMTP transaction occurring over a telnet connection, allowing you to see the results of each command being sent. Commands sent by the client are highlighted in **blue** and responses received from the server are in **red**.

An SMTP transaction being made using telnet:

```
[username@username-m0 ~]$ telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 username-m0.unix.useractive.com ESMTP Postfix
HELO localhost
250 username-m0.unix.useractive.com
MAIL FROM:username@localhost
250 2.1.0 Ok
RCPT TO:username@localhost
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From: username@localhost
To: username@localhost
Subject: I'm sending mail with SMTP commands over telnet

Hello. I used telnet to initiate an SMTP session with my local mail server and now I'm
sending you a message to let you know about it. Pretty cool, huh?
.
250 2.0.0 Ok: queued as 5CD445FAB
QUIT
221 2.0.0 Bye
Connection closed by foreign host.
```

Note

That special character sequence to end the DATA command, <CR><LF>.<CR><LF>, corresponds to pressing **Enter**, then **.**, then **Enter** again.

This is more or less what's going on behind the scenes when you send an email from your favorite mail client (though there are probably some commands related to SSL encryption and authentication as well). After receiving the message from the sender, the mail server then begins attempting to deliver it to the recipient. If the sender and the user reside on the same mail exchanger, then the process is essentially complete, and the message is delivered to the appropriate mailbox on the server. If the recipient does not exist on the same server, then the correct mail exchanger must be located and contacted. This process starts with the *MTA* or *Mail Transfer Agent* (this is the software that runs on your mail server) looking up the MX record for the domain in the recipient's email address. The MTA then initiates an SMTP

session with the mail exchanger listed in the MX record it just received via DNS. If this session completes successfully with the delivery of the message to the appropriate MX, the MTA will log the successful delivery and move on to the next message. However, if there is a problem with delivery, such as the mail exchanger for the recipient's domain being unavailable, the message then gets sent to a queue for redelivery at a later time. The delay between redelivery attempts is not standard, and is up to the server administrator to choose. Make a mental note about this particular feature of MTAs, as we will employ it later for a certain type of spam control.

Assuming everything has gone as it should so far with our theoretical message, the MTA for the recipient's domain should have the message now. On many mail servers, this isn't necessarily the end of the story. The final step of delivery is often put in the hands of a *MDA* or *Mail Delivery Agent*. MDAs take messages from the MTA software and "deliver" them to users' mailboxes, often doing filtering operations to allow mail messages to be sorted into folders within a user's mailbox. In some larger installations, the MTA will deliver the message to an external IMAP or POP server that actually handles mail retrieval sessions for clients.

Getting Familiar With Postfix

Luckily, your machine already has a mail transfer agent called *postfix* installed, so you don't need to install one yourself. The postfix MTA was designed to be a compatible replacement for the aging and difficult-to-configure sendmail. The postfix installation on your machine is actually already functional for some cases, but you must first make a small modification in order for it to send mail out of the Linux Learning Environment. Open the **/etc/postfix/main.cf** file for editing (you must use root privileges to do this), find the **relayhost** parameter, and add the following:

CODE TO TYPE:

```
#relayhost = $mydomain
#relayhost = [gateway.my.domain]
#relayhost = [mailserver.isp.tld]
#relayhost = uucphost
#relayhost = [an.ip.add.ress]
relayhost = 172.16.0.1
```

When you finish, write the file out. In order for the changes to take effect, you must cause postfix to reload its configuration file. You can do this using the **service** command with the **reload** subcommand:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo service postfix reload
Reloading postfix: [ OK ]
[username@username-m0 ~]$
```

Note

Using the **reload** subcommand rather than **restart** causes the service to reread its configuration file without stopping service.

Now your machine should be able to relay mail to the outside world. In order to test this, you can send yourself an email. For practice, try using telnet to manually issue the SMTP commands to your mail transfer agent. Follow the below example, substituting your personal email address for *your@email.address.com* and as usual, your username for *username*:

INTERACTIVE SESSION:

```
[username@username-m0 ~]# telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 username-m0.unix.useractive.com ESMTP Postfix
HELO localhost
250 username-m0.unix.useractive.com
MAIL FROM:<username@username-m0.unix.useractive.com>
250 2.1.0 Ok
RCPT TO:<your@email.address.com>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
To: your@email.address.com
From: username@username-m0.unix.useractive.com
Subject: My first email!

Testing one two three
<Enter>.<Enter>
250 2.0.0 Ok: queued as D5E762665
QUIT
221 2.0.0 Bye
Connection closed by foreign host.
```

Remember, you must press **<Enter>.<Enter>** to terminate the DATA command. Now go check to see if you've received the email you sent to your personal email account. You may have to wait a few minutes for it to appear, depending on the configuration of your personal email. Once you receive it, you have confirmed that your MTA is working properly for outgoing messages. However, it is not set up to receive email from remote hosts by default. This is done because most systems will not be tasked with relaying email. Allowing your server to accept incoming mail means changing the default postfix configuration slightly. Initially, your MTA is only configured to listen on "localhost," meaning it will not accept incoming connections from any other host. Obviously this is not desirable for receiving mail from other machines. Open the **main.cf** file again, find the **RECEIVING MAIL** section and locate the **inet_interfaces** parameter. To allow postfix to listen for remote connections, change the value from **localhost** to **all** as shown:

CODE TO TYPE:

```
#inet_interfaces = all
#inet_interfaces = $myhostname
#inet_interfaces = $myhostname, localhost
inet_interfaces = localhostall
```

You should now be set to receive email from a remote host! However, these changes won't take effect until you force postfix to reload its configuration. Do this using the service command with the **restart** subcommand (not the **reload** subcommand used before. This is required when changing the **inet_interfaces** parameter). Now try to send an email to "username@username-m0.unix.useractive.com" from your personal email account. You can use a mail reader program called **mutt** to verify that the email arrived. Go ahead and run **mutt** in your console. The first time you start mutt, you are asked if you want to create a Mail directory in your home directory. Press **Enter** here to say yes.

INTERACTIVE SESSION:

```
[username@username-m0 ~]# mutt
/home/username/Mail does not exist. Create it? ([yes]/no):
```

After that, you see mutt's user interface:

```
q:Quit d:Del u:Undel s:Save m:Mail r:Reply g:Group ?:Help
1 Apr 10 username@domain ( 1) A test email

---Mutt: /var/spool/mail/username [Msgs:1 0.8K]---(date/date)-----all)---
```

You may have more than one email in your inbox. If that's the case, you can use the arrow keys on your keyboard to highlight the message you want to view and then press **Enter** to display it. To exit from the message, press **q**, and to leave mutt altogether, press **q** again. You can now send mail to your machine, but there's a bit of a catch here. You may have noticed that you had to send email to "username@username-m0.unix.useractive.com". This isn't really what you want with your mailserver. Ideally, you'd like to be able to send email to "user@domain" rather than "user@host." In your case, this means sending mail to "user@username.unix.useractive.com". With the configuration you have in place right now, mail sent to "user@username.unix.useractive.com" will not be accepted for delivery on your mailserver, because it doesn't know it should accept mail for this domain. In order to fix this, you must edit a few parameters in `/etc/postfix/main.cf`. First are the **myhostname** and **mydomain** parameters.

CODE TO TYPE:

```
# INTERNET HOST AND DOMAIN NAMES
#
# The myhostname parameter specifies the internet hostname of this
# mail system. The default is to use the fully-qualified domain name
# from gethostname(). $myhostname is used as a default value for many
# other configuration parameters.
#
#myhostname = host.domain.tld
#myhostname = virtual.domain.tld
myhostname = mail.username.unix.useractive.com

# The mydomain parameter specifies the local internet domain name.
# The default is to use $myhostname minus the first component.
# $mydomain is used as a default value for many other configuration
# parameters.
#
# mydomain = domain.tld
mydomain = username.unix.useractive.com
```

The **myhostname** and **mydomain** parameters are used in many places throughout the main.cf file. Defining them here makes it easier to maintain the configuration because it means you won't have to change your host and domain name in several places in the file. Next, you can change the parameter that determines what hosts and domains your mailserver will deliver mail for, **mydestination**. You will want it to accept mail for localhost, the domain **username.unix.useractive.com**, the host **mail.username.unix.useractive.com** and the host **username-m0.unix.useractive.com**.

CODE TO TYPE:

```
#mydestination = $myhostname, localhost.$mydomain, localhost
#mydestination = $myhostname, localhost.$mydomain, localhost, $mydomain
#mydestination = $myhostname, localhost.$mydomain, localhost, $mydomain,
#      mail.$mydomain, www.$mydomain, ftp.$mydomain
mydestination = localhost, $myhostname, $mydomain, username-m0.unix.useractive.com
```

After saving the file, reload postfix's configuration:

INTERACTIVE SESSION:

```
[username@username-m0 ~]# sudo service postfix reload
Reloading postfix: [ OK ]
```

Now try sending mail to **username@username.unix.useractive.com**. You can use mutt to check for its arrival. If you don't receive it, contact your mentor for assistance. If you do, then you've successfully configured your mail server to receive mail for your domain! Continue on to learn about a handy feature called *mail aliases*!

Email Aliases

Email aliases are a useful way to change the final delivery destination of an email. Not only can you use them to forward email from one user to another, but you can also use them to send email to a specific file or even to a command! For now, we'll address using aliases to deliver mail destined for one user to another. We can easily demonstrate the power of aliases by creating one that will send mail from an arbitrary username to your username. For this, use your initials (if your name is Alan B. Cartman, use "abc"). Aliases are located in the **/etc/aliases** file. There is one alias per line. Open the aliases file for editing using root privileges, check for existing aliases that are the same as the one you want to use, and add a new alias for yourself at the bottom:

CODE TO TYPE:

```
# Person who should get root's mail
#root:      marc

initials:    username
```

After you save the file, you need to run the **newaliases** command in order to make postfix reread the aliases database. Again, this requires root privileges.

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo newaliases
[username@username-m0 ~]$
```

Now, send an email from your personal email account to the alias you just created. If your alias was "abc," send it to "abc@username.unix.useractive.com". You can then use mutt to see if you received the email in your inbox. Pretty cool, huh?

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Postfix: Security

In the previous lesson, you configured your installation of postfix to accept incoming mail and send outgoing mail. This is an important first step in configuring a mail server, but it doesn't end there. In this lesson (and the next) we'll cover topics about securing your mailserver against unauthorized use and spam.

An Introduction to Postfix Security

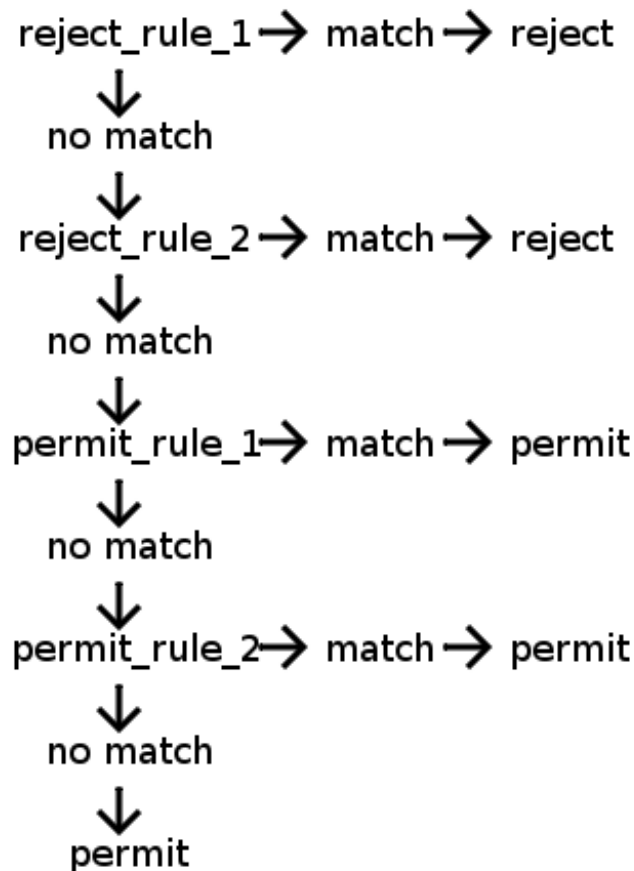
Most mail server security problems come down to one enemy: spam. While spam may seem like more of an inconvenience than a security issue, it's actually both. Security issues can arise from email that contains viruses, and there are also indirect consequences. For example, a scam or phishing email can convince a user to give up his or her personal information, including logins and passwords. While it seems this would only cause harm to the affected user, this kind of data in the hands of a hacker can give them a foothold to your system that could lead to a wider security breach. Do not underestimate the power of spam to not only annoy, but endanger your systems.

One of the first steps you can take to prevent spam is to configure your mail server to be selective about which email it will accept. A mail server that will accept or forward any email is called an *open relay*; you don't want that. Hackers use *bot-nets* to scan the internet for open mail relays. When they find one, they can instruct their spam-generating bots to direct their mail through the open relay, and make it appear to be from a legitimate mail server. Mail that appears to be from a legitimate mail server will defeat many spam countermeasures, and enable spam to reach its intended destination. If your mail server is acting as an open relay, it will be *blacklisted* by most large email providers. A site that blacklists your mail server will reject mail from you, and your users will be unable to send legitimate mail. Getting off of open relay blacklists takes quite a bit of time and effort, so it's best to avoid ending up there in the first place.

In addition to stopping the relay of spam, you want to prevent local delivery of spam. There are powerful tools available that scan incoming email messages and rank them based on their probability that they are spam. There are also tools that exploit some of the fundamental differences in the way that spam and legitimate emails are sent in order to sort the good from the bad. We'll cover these topics in the next lesson. For this lesson, we'll concentrate on controlling *relay access* on your server.

Controlling Relay Access

You can control relay access in postfix using a few different parameters; we'll focus on the **smtpd_recipient_restrictions** parameter. This parameter is used to set relay access restrictions and is required in order for postfix to function. Values assigned to this parameter form a chain of rules that are evaluated after the RCPT command is received in an SMTP session. The rules are processed using information that postfix has gathered from the HELO, MAIL, and RCPT commands, in addition to information about the session itself, such as the IP address from which the connection originated. The ordering of the rules is critical in order to avoid acting as an open relay unintentionally. Rules are processed in such a way that if a rule is matched, an action is taken based on that rule and no further rules are processed. Rules can either take the action of permitting a session, or rejecting it. If a particular session does not match any of the rules in the chain, the session is permitted. This flow chart shows that process:



Ordering rules is important. For example, you could place a rule that checks the supplied HELO value against a list of acceptable HELO values. If the value "localhost" was in your list, an attacker could configure a bot to use the value "localhost" for the HELO command and be granted relay access on your server immediately, even if that same session would be rejected later in the chain due to another rule.

With this information in mind, let's get to work securing your relay access. Before we write any rules for **smtpd_recipient_restrictions** though, we'll define the **mynetworks** parameter. We'll set it to a list of IP addresses and/or network addresses from which we'll always accept mail. Let's assume for now that in addition to the 172.16.0.0/12 network, your mail server has an address on a 192.168.0.0/24 network, and acts as the mail gateway for hosts located in that network. You don't actually want to accept mail from all clients on the 172.16.0.0/12 network, because there are machines on that network that you don't control. However, since you are the administrator for the theoretical 192.168.0.0/24 network, you're safe relaying all mail from there. Additionally, you want to accept all mail originating from the mailserver itself, 127.0.0.1 (localhost). So, define the **mynetworks** parameter in **/etc/postfix/main.cf** as shown:

CODE TO TYPE:

```
#mynetworks = 168.100.189.0/28, 127.0.0.0/8
#mynetworks = $config_directory/mynetworks
#mynetworks = hash:/etc/postfix/network_table
mynetworks = 127.0.0.1/32, 192.168.0.0/24
```

Note

When you specify multiple values for a parameter in the postfix configuration, separate them with commas. Multiple values can even span multiple lines in the file, as you'll see when we work on the **smtpd_recipient_restrictions** parameter.

Perfect! Load this new configuration by reloading postfix, then verify that the settings have been updated using the **postconf** tool. This prints the currently in-use configuration of postfix. Running **postconf** without any arguments prints the whole configuration, whereas running it with a parameter name as an argument will print that specific parameter.

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo service postfix reload
Reloading postfix: [ OK ]
[username@username-m0 ~]$ postconf mynetworks
mynetworks = 127.0.0.1/32, 192.168.0.0/24
```

If you get the same output, then you've made the correct changes. If you don't, check the configuration file for typos and be sure to reload the configuration. Once you have your **mynetworks** parameter properly defined, you can begin defining permit and reject rules for the **smtpd_recipient_restrictions** parameter. This parameter is not defined in your **main.cf** file currently, so you'll need to add it at the end. Open **/etc/postfix/main.cf** to edit and insert a new line at the bottom:

CODE TO TYPE: Your first smtpd_recipient_restrictions:

```
smtpd_recipient_restrictions = permit_mynetworks,
                               reject
```

Write the file out, reload the configuration, and verify that the parameter has changed:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo service postfix reload
Reloading postfix: [ OK ]
[username@username-m0 ~]$ postconf smtpd_recipient_restrictions
smtpd_recipient_restrictions = permit_mynetworks, reject
```

Excellent! This brings us to an interesting point. We've set the **smtpd_recipient_restrictions** parameter, but permitting mail to be *sent* from **mynetworks** seems to be a restriction on the sender. That's kind of odd. Here's what's actually happening: all machines on our theoretical local network (192.168.0.0/24) can specify any recipient, and their mail will be accepted. Giving one or more clients unrestricted access to relay mail through your server is called *whitelisting*. Be careful with your whitelisting powers, and err on the side of more restrictive policy. The rule set you have so far would be fine if you were sending email only from local senders to remote destinations, however it doesn't allow anyone to send mail to your server for local delivery. This is a problem.

Using a blanket "reject" at the end of this chain of rules means that a session that doesn't originate from one of **mynetworks** will be rejected. Email originating from remote hosts will not satisfy that rule, and will be rejected. We need a reject rule to reject certain email. In this case, the **reject_unauth_destination** rule works well. The rule instructs postfix to reject any email in which the recipient's domain is not defined in either the **relay_domains** parameter or the **mydestination** parameter. You may recall from the last lesson that the **mydestination** parameter is set to a list of host and domain names for which postfix will execute final delivery. An email's recipient address must have a domain that is listed in the **mydestination** parameter, or the session will be rejected. Go ahead and replace the **reject** rule with the more reasonable **reject_unauth_destination** rule:

CODE TO TYPE:

```
smtpd_recipient_restrictions = permit_mynetworks,
                               reject
                               reject_unauth_destination
```

Let's break down the current relay access situation. Anybody listed in the **mynetworks** parameter is allowed to relay mail to anywhere, including local and remote locations. Mail originating from remote locations can be relayed for internal delivery only. Mail originating from remote locations and destined for remote locations will not be relayed, because there are no **relay_domains** defined in your configuration. It seems like we've arrived at a pretty secure state, doesn't it? In reality, this is actually the default configuration for postfix when no rules are specified for **smtpd_recipient_restrictions**. It works fairly well when users plan to send email only from your theoretical private network, but it prevents some common real-world use cases, like when one of your users tries to send email through your mail relay from outside of the office. One way to overcome this is to enable SMTP authentication, which allows remote users to supply a password to gain relay access. Unfortunately, due to the limitations of the Linux Learning Environment, we can't demonstrate this to you. If you are interested in learning about SMTP authentication, you can find more information [here](#). Before we move on, make sure to reload your postfix configuration so you can send mail to your server from remote locations.

Greylisting: Another Method For Relay Access Control

So far, we've discussed methods for controlling relay access that passively use data collected from the SMTP session. There's another method you can use in conjunction with the methods we've discussed already that gives the mail server a more active role in rejecting relay access for bogus email. You may recall from the previous lesson that one feature of SMTP is that on a properly configured SMTP server, messages that are not delivered successfully are queued up for later redelivery. We can exploit this feature to filter out many spam bots that are not attempting redelivery, using a method called *greylisting*. Most spam bots are not written to be fully featured SMTP clients, and they usually lack features such as queueing *bounced messages* for later redelivery (bounced messages are messages that are not accepted for delivery, but instead remain on the originating mail exchanger). If the spam bot in question did work on redelivery, it would likely spend more time trying to redeliver bounced messages than sending out new spam. Legitimate mail servers will attempt redelivery.

Greylisting works by checking the sender/recipient address pair from the SMTP session. If the greylisting daemon has not seen that specific pair of addresses before, it instructs postfix to bounce the message, and makes a note of the message in its database. If a spam bot that doesn't do redelivery sent the message, the message disappears. However, if the message originated from a legitimate mail server, redelivery will be attempted later, and the second time the greylisting daemon sees the message, it will accept it for relay or delivery. The daemon then whitelists the sender/recipient pair in its database so that subsequent messages with that pair will be let through on the first attempt. Install the **postgrey** package using **yum**, then use **chkconfig** to enable it at boot time and **service** to start it:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum install postgrey
... lots of yum output...
[username@username-m0 ~]$ sudo chkconfig postgrey on
[username@username-m0 ~]$ sudo service postgrey start
```

Once you have postgrey enabled and running, you can instruct postfix to use postgrey to check incoming mail. This can be done using the postfix parameter **smtpd_recipient_restrictions**. Open **/etc/postfix/main.cf** to edit, locate **smtpd_recipient_restrictions**, and add these restrictions (don't forget to add the comma after **reject_unauth_destination**):

CODE TO TYPE:

```
smtpd_recipient_restrictions = permit_mynetworks,
                               reject_unauth_destination,
                               check_policy_service unix:/var/spool/postfix/postgrey/socket
```

The "check_policy_service" directive does exactly that; it tells postfix to check the email message against a service that's responsible for policy decisions. The value for this parameter is set to a socket where the policy service daemon is listening. In this case, the policy service daemon is listening on a *unix socket* (which is a special kind of file that processes can use to communicate with each other) located at **/var/spool/postfix/postgrey/socket**. This is the default location for postgrey to listen. Now, reload your postfix configuration so it starts executing greylist checks. In order to see if it's working, view your mail log by running **sudo tail -f /var/log/maillog** (you may remember tail from our first course in this series).

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo tail -f /var/log/maillog
... output from the maillog ...
```

Now for the big test! Try to send an email to **username@username.unix.useractive.com** from your personal email address. If greylist is working, you'll see a message like this scroll onto the console where you're following the maillog:

OBSERVE:

```
Apr 13 15:27:45 localhost postgrey[1978]: action=greylist, reason=new, client_name=unknown, client_address=172.16.0.2, sender=your@email.address.com, recipient=username@username.unix.useractive.com
Apr 13 15:27:45 localhost postfix/smtpd[2141]: NOQUEUE: reject: RCPT from unknown[172.16.0.2]: 450 4.2.0 <username@username.unix.useractive.com>: Recipient address rejected: Greylisted, see http://postgrey.schweikert.ch/help/username.unix.useractive.com.html; from=<your@email.address.com> to=<username@username.unix.useractive.com> proto=ESMTP helo=<tequila.useractive.com>
```

Notice the reason for rejection: "Recipient address rejected: Greylisted." This tells you that greylisting is in effect. Your email has been rejected temporarily by your mailserver. If you wait patiently, another attempt to deliver your email will be made, and this time it will succeed. The amount of time until redelivery depends on how the mail exchanger that handles your personal email is configured. Continue to follow your maillog to see when the redelivery attempt is made.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

System Logs

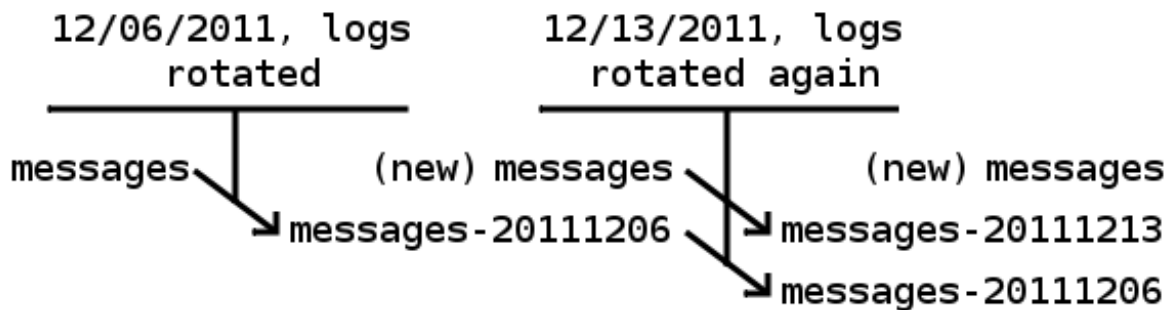
In this lesson, you will learn about the all-important *system logs*. System logs record all sorts of useful data about your system as it operates.

What the Logs Tell You

The system logs are located in **/var/log**. Take a moment to `cd` into that directory and look around. You'll see several files with names like, "messages" and "secure." Each log file is configured to receive specific information about the operation of your system. The "secure" log, for example, has a list of events related to logins as well as sudo actions. Here are the most common log files and their functions:

cron	The cron log keeps track of all of the cron events that occur on the system. Cron is used to execute commands and scripts automatically at set times.
dmesg	The dmesg log contains messages from the kernel, including boot-time messages and messages related to kernel activities like device discovery.
maillog	The maillog file keeps track of mail messages that the system has processed, as well as errors encountered by the mail server. Each entry contains information about the sender, recipient, time the message was processed, and the status of the message. It does not include the contents of the message.
messages	The messages log is where general system messages go. It's kind of a catch-all for logged events. Some events will go to the messages log as well as another log. If there's a problem with the system, this is often the first place to look for more clues as to what might be failing.
secure	The secure log holds information about all system logins, successful or failed. It also keeps track of all sudo activity, including the user who attempted to sudo and the command that was being run with sudo.

There are several versions of each file in **/var/log**, with names like, "messages-20111206." There is a program named *logrotate* that keeps the log files under control. Logrotate decides whether to rotate a particular log, based on the size of that log file. When a predefined size limit is reached, logrotate processes that particular log file. Each time logrotate operates on a log, the current log file is renamed using the current date and a new log file is created:



Logrotate is configured so that when there are four archived log files and another rotate occurs, the oldest log file is removed from the system. This keeps the number of log files (and the space that they take up) in check.

Ways Of Viewing Log Files

Most log files can be viewed directly using `cat`, `less`, `vi`, or any of several other methods. **tail** and **grep** are well-suited for viewing log files. **Tail** and **head**, print out the last or first ten lines of a file, respectively. **Head** is usually not too useful for looking at log files, so we'll focus on **tail**. Tail gives you a quick way to look at the most recent activity in a log file. Let's tail the secure log:

INTERACTIVE SESSION:

```
[username@username-m0 log]$ sudo tail /var/log/secure
Jan 12 16:21:08 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail /var/log/messages
Jan 12 16:21:09 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail /var/log/messages
Jan 12 16:21:19 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail /var/log/messages
Jan 12 16:22:17 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail /var/log/messages-20111206
Jan 12 16:22:31 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail /var/log/messages-20111211
Jan 12 16:22:45 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail secure
Jan 13 14:57:11 username-m0 sudo: pam_unix(sudo:auth): authentication failure; logname=
username uid=0 euid=0 tty=/dev/tty0 ruser=username rhost=username-m0.unix.useractive.co
m user=username
Jan 13 14:57:12 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/bin/bash
Jan 13 14:57:15 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail maillog
Jan 13 14:57:22 username-m0 sudo: username : TTY=tty0 ; PWD=/var/log ; USER=root ; COMM
AND=/usr/bin/tail secure
```

You'll see something like the log above. You can change the number of lines that tail prints out by specifying **-n number**. Tail has another useful feature called *following*. By running tail with the **-f** option, tail will continue to print lines from the file as they are written. Try tailing secure again, but this time with the **-f** option. Tail will keep control of the terminal rather than returning it to the shell. Switch over to another one of your consoles and log in as testuser. When you switch back to the console where tail is running, you'll see an entry that corresponds to jsmith logging in. You can use the follow feature to watch a log file as events are occurring on the system. This can help you to find and fix problems that you might not catch otherwise. Following a log file with "tail -f" can be especially powerful when used in conjunction with grep. Piping the output of "tail -f" to grep allows you to filter events in real time. In a "busy" log file like the mail log on a mail server, this can be tremendously helpful.

Grep, More In Depth

Earlier in the course series, we introduced the grep command as a way to search for strings in a file or stream of data. However, there is a large range of features that grep offers. In addition to basic ways to modify your search, grep can search for patterns of characters specified using *regular expressions*.

Let's look at a couple of ways you can modify a grep search to make it more useful. Grep searches, by default, are *case-sensitive*, but you can instruct grep to do a case-insensitive search with the **-i** flag. To see this in action, try these commands:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ grep iana /etc/services
#      http://www.iana.org/assignments/port-numbers
[username@username-m0 ~]$ grep -i iana /etc/services
# IANA services version: last updated 2009-11-10
# Note that it is presently the policy of IANA to assign a single well-known
# The latest IANA port assignments can be gotten from
#      http://www.iana.org/assignments/port-numbers
#>The Registered Ports are listed by the IANA and on most systems can be
#>The IANA registers uses of these ports as a convenience to the
# Kerberos 5 services, also not registered with IANA
# Updated additional list from IANA with all missing services 04/07/2008
[username@username-m0 ~]$
```

When you want to remove information that you *don't* want to see, rather than trying to build a search pattern that will return all the information you *do* want to see, use *inverse searching*. Inverse searching takes the string or regular

expression you supply to `grep`, and returns everything that *doesn't* match that string or expression. For example, you can find all of the lines in `/etc/services` that don't include the letter "a" by doing `grep -v a /etc/services`.

Another important flag for controlling how `grep` matches things is `-w`. By default, `grep` considers a pattern to match a line if that pattern appears anywhere in that line. This means that if your pattern matches a whole word, part of a word or anything else, it's a match. The `-w` flag forces `grep` to interpret your pattern as a full word, not just part of a word. For example, `grep -w book` would match "book," but not "bookkeeper" or "cookbook."

An Introduction to Regular Expressions

Regular expressions allow you to match strings based on a set of criteria, rather than matching against one specific string. Suppose you want to return all events from a log file that occurred between 12:00:00 and 12:59:59. Grepping for "12" would return events from that range, but it would also return events occurring when the minute, second, or any other part of the line included "12." Use regular expressions to restrict the matches.

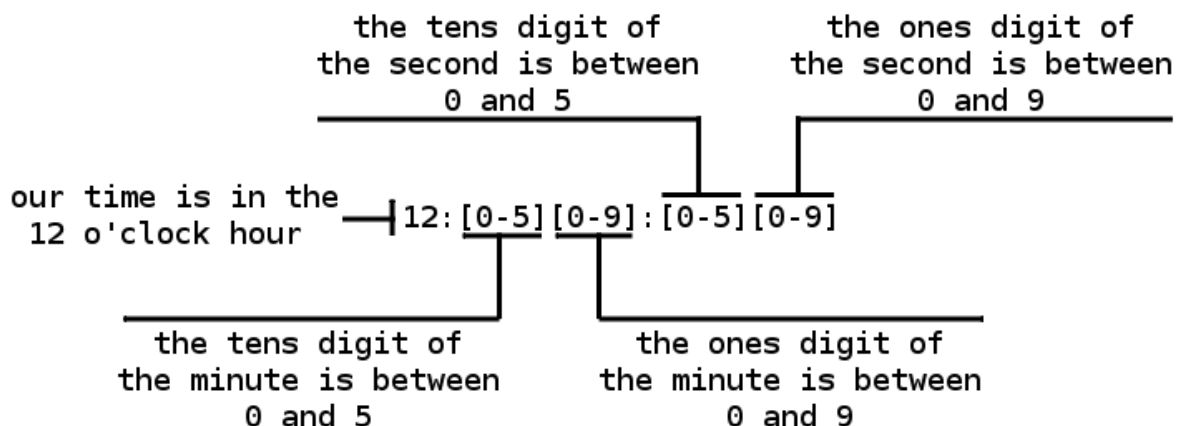
The most basic unit in regular expressions is the period (.) character. The period matches any single character, including all letters, digits, white space, or special characters. Let's try a few examples:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ echo "anz" | grep a.z
anz
[username@username-m0 ~]$ echo "a?z" | grep a.z
a?z
[username@username-m0 ~]$ echo "a z" | grep a.z
a z
[username@username-m0 ~]$ echo "az" | grep a.z
[username@username-m0 ~]$ echo "amnz" | grep a.z
[username@username-m0 ~]$
```

Consider our 12 o'clock problem again. Assuming your time will always be listed in the format "HH:MM:SS," you could `grep` for "12:...," right? You could *almost* be assured that all of the lines that matched that expression would have a timestamp between 12:00:00 and 12:59:59. While it's a stretch, it's possible that you would match other lines from the log file because you haven't specified that the MM and SS portions of the pattern should be numbers between 00 and 59. You might return something that included a larger number in one of the other fields, or even letters. This is where *bracket expressions* come in handy.

Bracket expressions allow you to be more specific with matching by giving you the power to specify a set or range of characters that are allowed in a particular position. This list or range is written between square brackets ([]). For example, you could match all vowels by using [aeiou], or you could match any single digit between one and five by using [1-5]. A bracket expression that solves our problem would look like "12:[0-5][0-9]:[0-5][0-9]":



Another method you can use to narrow your focus is *anchoring*. Anchoring allows you to specify the location of your pattern in your line. The anchoring characters (which you may recognize from the Text Editors lesson) are `^` and `$`; they indicate the beginning and end of the line, respectively. The pattern `^we` would match "we were there", but not "there we were", because the `^` requires "we" to be at the beginning of the line. To use `$` to match strings at the end of the line, place it at the end of the pattern, like `there$`. This pattern would match "we

were there," but not "there we were." Here are some examples of simple regular expressions:

b..k	Matches "book," "back," and "beak," but not "break".
week.end	Matches "week end" and "week-end", but not "weekend."
too[a-m]	Matches "took" and "tool," but not "cartoon."
to[^px]ic	Matches "tonic," but not "toxic" or "topic."
^hello	Matches "hello world!", but not "hi, hello!"
goodbye\$	Matches "goodnight and goodbye." but not "goodbye everybody!"

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

PHP

So far, you've installed and configured Apache's httpd, which is great if all you plan to do is produce plain HTML websites. However, many locations require dynamic content, and you'll want to use PHP to generate it. Apache's httpd does not have the ability to parse PHP files and generate web pages by default, but it is possible to add this functionality by installing a PHP module. In this lesson we'll go through the process of building and installing a PHP module for your httpd installation.

Installing PHP

As with Apache's httpd, the first step in the process is to download the source code for PHP. Here, you can use **wget** to download the source to your home directory from the HTTP mirror located at **<http://us.php.net/get/php-5.4.24.tar.bz2/from/this/mirror>**.

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ wget -O http://us.php.net/get/php-5.4.24.tar.bz2/from/this/mirror
--2012-04-09 11:36:37-- http://us.php.net/get/php-5.4.24.tar.bz2/from/this/mirror
Resolving us.php.net... 208.69.120.58
Connecting to us.php.net|208.69.120.58|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://us.php.net/distributions/php-5.4.24.tar.bz2 [following]
--2012-04-09 11:36:37-- http://us.php.net/distributions/php-5.4.24.tar.bz2
Connecting to us.php.net|208.69.120.58|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11439508 (11M) [application/octet-stream]
Saving to: `php-5.4.24.tar.bz2'

100%[=====>] 11,439,508  3.71M
/s   in 2.9s

2012-04-09 11:36:40 (3.71 MB/s) - `php-5.4.24.tar.bz2' saved [11439508/11439508]
```

Wget allows you to retrieve web pages or files using an HTTP request. Once the source is downloaded, you must unarchive it. To do that, you'll use the same procedure you used for unarchiving the httpd source code:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ tar xvjf php-5.4.24.tar.bz2
... lots of output ...
php-5.4.24/build/scan_makefile_in.awk
php-5.4.24/build/shtool
php-5.4.24/autom4te.cache/output.0
php-5.4.24/autom4te.cache/requests
php-5.4.24/autom4te.cache/traces.0
[username@username-m0 ~]$
```

The process of configuring and building PHP is similar to the process used to build httpd, but with one key difference. The PHP build process needs to know where a particular part of httpd, called apxs (the **ap**ache **ex**ension **tool**), is located, so it can use it to build a compatible module. Apxs is a **binary** tool that is part of the httpd installation you did at **/usr/local/apache2**. It's located in **/usr/local/apache2/bin**. You will supply this information to PHP's configure script. Change directory into the PHP source directory you just unarchived and run the configure script with the flag **--with-apxs2=/path/to/the/apxs/binary** (substituting the appropriate path, of course). The configure script will take a while to complete, probably about 10 minutes:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ cd php-5.4.24
[username@username-m0 php-5.4.24]$ ./configure --with-apxs2=/usr/local/apache2/bin/apxs
... lots of output ...
config.status: creating scripts/php-config
config.status: creating scripts/man1/php-config.1
config.status: creating sapi/cli/php.1
config.status: creating main/php_config.h
config.status: executing default commands
[username@username-m0 php-5.4.24]$
```

Once this is complete, you can build and install the generated *shared library*. A shared library is a blob of code that contains functions that can be used by other binaries to add features. In this case, the feature is the ability to parse PHP code and use it to generate HTML, which you can then send to the client's browser. The build process for php will take close to an hour to complete, so after you start the build, feel free to get up and stretch your legs! When the build is done, copy the file **php.ini-production** to **/usr/local/lib/php.ini**. This is PHP's global configuration file.

INTERACTIVE SESSION:

```
[username@username-m0 php-5.4.24]$ make
... lots of output ...
[username@username-m0 php-5.4.24]$ sudo make install
... a little more output ...
[username@username-m0 php-5.4.24]$ sudo cp php.ini-production /usr/local/lib/php.ini
```

A PHP shared library is installed in the **/modules** directory inside httpd's server root. The install process also modifies your server's **httpd.conf** file to load the module, which makes PHP processing available. However, the install process does not add the files to **httpd.conf** that instruct the server which files to process using the PHP interpreter. You can enable the use of PHP using virtual host (you don't want to allow users to host arbitrary PHP scripts out of their home directories). You'll need to enable apache to handle PHP files manually. Do this by associating a particular *MIME type* with php files. Httpd uses MIME type associations to determine how to interpret files based on their filename extensions. For PHP files, the correct MIME type is **application/x-httpd-php**. You can use the **AddType** directive to define this MIME type for php files. Go ahead and do this for the **www0** virtual host by adding the **AddType** directive to your **www0.conf** file as shown:

CODE TO TYPE:

```
<VirtualHost *:80>
    ServerName www0.dbasset1.unix.useractive.com
    ServerAlias www0
    DocumentRoot /var/www/www0
    AddType application/x-httpd-php .php

    <Directory "/var/www/www0">
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Restart httpd gracefully for the changes to take effect (**/usr/local/apache2/bin/apachectl graceful**). Now you can create a test script to see if PHP is working. Open the **/var/www/www0/hello.php** file and add this:

CODE TO TYPE: A Basic PHP Test Script:

```
<?php
    print "Hello world!";
?>
```

Load **http://www0.username.unix.useractive.com/hello.php** in your web browser. You'll see "Hello world!" Before we finish, let's demonstrate that we have enabled PHP only for the **www0** virtual host, and no other virtual hosts on your machine. Copy the file **hello.php** from the document root for **www0** into the document root of **www1**:

INTERACTIVE SESSION:

```
[username@username-m0 www0]$ cp hello.php ../www1/
```

Load **<http://www1.username.unix.useractive.com/hello.php>** in your browser. You see the contents of the script itself, rather than the output "Hello world!" because we haven't associated a MIME type with the .php file extension for this particular virtual host (www1) yet. As a result, httpd doesn't know how to handle the file and instead interprets it as plain text. You can see this in the full headers in the response to an HTTP GET request for hello.php from www1.username.unix.useractive.com (important data is **highlighted** below):

INTERACTIVE SESSION:

```
[username@username-m0 www0]$ telnet www1.username.unix.useractive.com 80
Trying 172.16.101.152...
Connected to www1.username1.unix.useractive.com.
Escape character is '^]'.
GET /hello.php HTTP/1.1
Host: www1.username.unix.useractive.com

HTTP/1.1 200 OK
Date: Mon, 09 Apr 2012 20:33:38 GMT
Server: Apache/2.2.22 (Unix) PHP/5.4.0
Last-Modified: Mon, 09 Apr 2012 20:31:15 GMT
ETag: "a709-1d-4bd44e34379e5"
Accept-Ranges: bytes
Content-Length: 29
Content-Type: text/plain

<?php
    print "Hello world!";
?>
```

On the other hand, a request for hello.php is returned as html...

INTERACTIVE SESSION:

```
[username@username-m0 www0]$ telnet www0.username.unix.useractive.com 80
Trying 172.16.101.152...
Connected to www0.username.unix.useractive.com.
Escape character is '^]'.
GET /hello.php HTTP/1.1
Host: www0.username.unix.useractive.com

HTTP/1.1 200 OK
Date: Mon, 09 Apr 2012 20:36:51 GMT
Server: Apache/2.2.22 (Unix) PHP/5.4.24
X-Powered-By: PHP/5.4.24
Content-Length: 12
Content-Type: text/html

Hello world!
```

That's it! You've installed PHP on your machine and you can now serve dynamic web content. In your final project, we'll tie together the lessons you've learned so far in this course about DNS, mail, *and* web servers. Good luck!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.
