Pro Nagios 2.0

James Turnbull

Apress®

Pro Nagios 2.0

ISBN-13: 978-1-59059-609-8

ISBN-10: 1-59059-609-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Sumser
Technical Reviewer: Justin Kulikowski
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Nancy Sixsmith
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 2

Basic Object Configuration

This chapter will get you started with the basic configuration of your Nagios server to provide monitoring for hosts and services. I'll begin by describing how Nagios works, move on to how it is configured, and then demonstrate how to create the basic configuration objects needed for monitoring your environment. I'll cover a number of topics in this chapter that form the basis for the principles and methods that Nagios uses to monitor your hosts and services. Some of these topics I'll cover in detail, but others I'll just introduce you to and then address them in greater detail in later chapters.

As a result of this process of gradual introduction, especially to the more advanced Nagios topics, I recommend you also read and use the Nagios documentation. The Nagios documentation is both expansive and highly detailed. It is a useful resource during the configuration process. The Nagios documentation is provided in HTML form with your Nagios installation and is viewable from the web console. The documentation is also available online at `http://nagios.sourceforge.net/docs/2_0/`.

How Does Nagios Work?

Let's quickly look at how the basics of Nagios work, starting with how to define your monitored environment. You do this by defining a series of objects that represent the characteristics of the environment being monitored. You begin by defining your assets to the Nagios server. Nagios calls them *hosts*. Then you define the attributes and functions of these assets. Nagios calls these *services*. Services can include attributes and functions such as an FTP daemon, an email server, an application, or a database. Services can also include the attributes of a host or application—for example, the amount of disk space free on a host or the number of transactions processed by an application. You also need to define the people who manage these assets and how they are contacted. These are the people who need to be notified if an event occurs on a host or service. Nagios calls these people *contacts*.

Once you have defined your assets, their attributes, and your people to Nagios, then you define the mechanisms for monitoring these assets or hosts and services. Nagios calls these mechanisms *commands*. Commands generally use a plug-in (which we installed in Chapter 1), binary, or script to check a service or host and return its state and status.

The act of monitoring hosts and services using a command is called a *check*. A check does two things. First, it returns the results of its checking to the server—for example, if you were monitoring disk space, then it would return the percentage of disk space free. Second, the check results are used to determine the status of the host or service. If the results of the check vary

from the conditions you have set, then the status of the host or service could change; if disk space reaches above a threshold you have set, for instance, the status of a service could change.

This change usually triggers a *notification*. This is a message from Nagios telling you about the change in your host or service. Notifications can be sent via a number of means: via email or Short Message Service (SMS), or you can customize Nagios to send via a variety of other mediums, such as to an Instant Messenger client. These notifications are sent to destination addresses configured in your contact objects and are sent using a particular type of command called a notification command.

So let's look at an example. I have a host called `kitten.yourdomain.com` that runs my company's email and web services, including Postfix, Apache, FTP, and DNS daemons. This host is managed by John, who works for my company's Unix Support team. John carries his Blackberry everywhere with him and is notified of events on the `kitten` host via emails to his Blackberry. John wants to know when any of the services on the `kitten` host are not responding and when the disk space on the host reaches 95 percent.

So how do I set up this monitoring? Well, first I must define a host object to Nagios that will specify the name and IP address of the `kitten` host. Next I define the services (through service objects) I wish to monitor on that host and specify which commands to use to monitor them. I include in the definition of the services being monitored when Nagios should check each service. I also define the conditions to monitor for, when and how often to trigger a notification, and whom to notify. For example, I create a service for the Postfix daemon on the `kitten` host. I tell Nagios that if it can't connect to the Postfix server, it should send a notification. Finally, I define a contact group and a contact object for John; the latter includes details of his email address that Nagios will use to send notifications.

I then start the Nagios server and it will begin monitoring the `kitten` host. If it detects any of the conditions I've defined, it will notify John so he can investigate and fix the problem.

In this chapter, I'll demonstrate how to create all the objects required for an example such as this.

How Is Nagios Configured?

The Nagios server, the web console, and your monitoring configuration are controlled by a series of configuration files that are usually located in the `/usr/local/nagios/etc` directory. There are three types of configuration files. The first are the Nagios server and web console configuration files, which handle the behavior of the server and web console. These files are named `nagios.cfg` and `cgi.cfg`, respectively. I'll look at some of the options in these files that support your monitoring configuration in this chapter. In later chapters I'll go into some of the other options contained in these files in greater depth.

■ **Tip** I recommend you read through the sample `nagios.cfg` and `cgi.cfg` files as they are well documented and will offer considerable insight into the functioning of Nagios.

Resource files are the second type of configuration files. They are designed to hold sensitive information, like database connection settings or macros that you want to hide from general view—for example, preventing them from being seen from the web console. I'll discuss them briefly later in the "Defining Commands" section.

Object configuration files are the third type of files. Objects include hosts, services, commands, and other types of objects that can be defined to Nagios. Objects represent the core of your Nagios configuration. Table 2-1 defines all the object types you can use with Nagios and what they do.

Table 2-1. *Nagios Object Types*

Object	Description
`host`	Hosts are physical devices like servers, routers, and firewalls.
`hostgroup`	Host groups are collections of hosts that generally have something in common, like their type or location.
`service`	Services run on hosts and can include actual services like SMTP or HTTP, or metrics such as disk space.
`servicegroup`	Service groups are collections of services that generally have something in common.
`contact`	Contacts are escalation or notification points that can potentially be contacted when an event occurs.
`contactgroup`	Contact groups are collections of contacts. All contacts need to be in a contact group.
`timeperiod`	Time periods are defined windows of time—for example, during business hours.
`command`	Commands are called by the check process to perform an action—for example, a command to check the status of a host using the `ping` command.
`servicedependency`	Allows a service or services to be dependent on other services.
`serviceescalation`	Provides a notification escalation process for services.
`hostdependency`	Allows a host or hosts to be dependent on other hosts.
`hostescalation`	Provides a notification escalation process for hosts.
`hostextinfo`	Host Extended Information changes and customizes the way hosts are displayed on the Nagios web console.
`serviceextinfo`	Service Extended Information changes and customizes the way services are displayed on the Nagios web console.

In Table 2-1, I've referred to the object types by the names used in the configuration files—for example, `hostgroup`. In this chapter, I'll interchangeably use this object name and the expanded name, host group. The same logic will apply to other object types. In this chapter I'll be principally looking at hosts and host groups, services and service groups, contact and contact groups, time periods, and commands. In subsequent chapters, I'll cover the other object definitions.

■ **Tip** This book principally focuses on new installations of Nagios 2.0. But if you're upgrading from a previous version of Nagios, some of the object configuration syntax has changed. I recommend you read the Nagios What's New page at `http://nagios.sourceforge.net/docs/2_0/whatsnew.html`. Additionally, there are some simple (use at your own risk) tools for migrating your configuration from Nagios 1.2 to 2.0 at `http://oss.op5.se/nagios/`.

Getting Started with Your Configuration

In order to get started with your Nagios configuration, it is useful to have a basis and a reference to build from. The sample configuration files you've installed provide an excellent basis for building your own configuration, and I highly recommend you use them.

If you installed Nagios from source, you will have installed the sample configuration files. These files are usually installed into the `/usr/local/nagios/etc` directory. All the installed files have a suffix of `-sample` on the end of the file. In order to use these files, you should remove the `-sample` suffix from the file. You can do that with this simple Bash script:

```
puppy# cd /usr/local/nagios/etc
puppy# for i in *-sample; do mv "$i" "${i%*-sample}";done
```

Two particular sample configuration files, `minimal.cfg` and `bigger.cfg`, contain examples of all the possible object types you can define to Nagios. The `minimal.cfg` configuration file is a very simple example of a Nagios configuration file. The `bigger.cfg` configuration file contains a considerably more detailed Nagios configuration.

■ **Tip** If you installed Nagios from an RPM, then you should already have the correctly named configuration files contained in the `/etc/nagios` directory. The files will have the `-sample` suffix already removed.

Specifying Your Configuration Files

So that the Nagios server knows about your configuration, you must define your configuration files to the Nagios server. To do this, you specify your configuration files and their location in the `nagios.cfg` file. To define the configuration files to the Nagios server, use the `cfg_file` directive. Nagios server configuration directives are divided into two parts: the directive name and the directive setting. These are separated by the = symbol like so:

```
directive=setting
```

Each directive should be on a new line. You can see an example of a Nagios directive here:

```
log_file=/usr/local/nagios/var/nagios.log
```

■ **Tip** Directives can be commented out of the `nagios.cfg` configuration file by prefixing the line with a # symbol. You can also use a # symbol to add comments to your configuration file. The sample `nagios.cfg` file is heavily commented.

The `cfg_file` directive tells Nagios the name and location of your configuration files. The setting is the full path to the configuration file being defined. In Example 2-1, you can see the `cfg_file` section of the sample `nagios.cfg` file.

Example 2-1. *The cfg_file Directive in nagios.cfg*

```
cfg_file=/usr/local/nagios/etc/checkcommands.cfg
cfg_file=/usr/local/nagios/etc/misccommands.cfg
cfg_file=/usr/local/nagios/etc/minimal.cfg
#cfg_file=/usr/local/nagios/etc/contactgroups.cfg
#cfg_file=/usr/local/nagios/etc/contacts.cfg
#cfg_file=/usr/local/nagios/etc/dependencies.cfg
#cfg_file=/usr/local/nagios/etc/escalations.cfg
#cfg_file=/usr/local/nagios/etc/hostgroups.cfg
#cfg_file=/usr/local/nagios/etc/hosts.cfg
#cfg_file=/usr/local/nagios/etc/services.cfg
#cfg_file=/usr/local/nagios/etc/timeperiods.cfg
#cfg_file=/usr/local/nagios/etc/hostextinfo.cfg
#cfg_file=/usr/local/nagios/etc/serviceextinfo.cfg
```

In Example 2-1, a number of configuration files have been defined but only three of these files—`checkcommands.cfg`, `misccommands.cfg`, and `minimal.cfg`—are not commented out. I'll come back to the first two files later in this chapter, but the third file, `minimal.cfg`, is a sample configuration file that contains examples of all of the available object types, combined into one file.

You can define as many configuration files as you like to Nagios. Any files you do define, however, must exist; otherwise, the Nagios server will return an error and fail to start. The data in your configuration files also must be valid as it is parsed by the Nagios server prior to the server starting. If the server detects invalid syntax or an illogical or invalid configuration, the Nagios server will fail to start.

■ **Note** I'll look at starting and stopping the Nagios server in Chapter 3.

How you group and store your object definitions in your configuration files greatly depends on your requirements. You can specify each object type in a separate file—for example, list all of your host objects in one file. You can also combine all types of objects in a single file, as is done in the sample configuration file, `minimal.cfg`. Or you could choose a different model and define all objects from a particular site or geographical location in separate files, as I've done here:

```
cfg_file=/usr/local/nagios/etc/india.cfg
cfg_file=/usr/local/nagios/etc/california.cfg
cfg_file=/usr/local/nagios/etc/australia.cfg
```

The `cfg_file` variables in Example 2-1 that are commented out reflect a configuration model where each object type is contained in a separate file. Throughout this book I'll use this model of configuration. Each object type will have its own separate file—for example, all the hosts contained in one file, `hosts.cfg`, and all the contacts contained in another file, `contacts.cfg`. In Example 2-2 I've commented out the sample configuration file (you could also simply delete the line) and uncommented the configuration files I'll be using in this chapter. Later I'll add other configuration files as we explore the remaining object types.

Example 2-2. *Individual Object Types Defined in Separate Files*

```
cfg_file=/usr/local/nagios/etc/checkcommands.cfg
cfg_file=/usr/local/nagios/etc/misccommands.cfg
#cfg_file=/usr/local/nagios/etc/minimal.cfg
cfg_file=/usr/local/nagios/etc/contactgroups.cfg
cfg_file=/usr/local/nagios/etc/contacts.cfg
#cfg_file=/usr/local/nagios/etc/dependencies.cfg
#cfg_file=/usr/local/nagios/etc/escalations.cfg
cfg_file=/usr/local/nagios/etc/hostgroups.cfg
cfg_file=/usr/local/nagios/etc/hosts.cfg
cfg_file=/usr/local/nagios/etc/services.cfg
cfg_file=/usr/local/nagios/etc/servicegroups.cfg
cfg_file=/usr/local/nagios/etc/timeperiods.cfg
#cfg_file=/usr/local/nagios/etc/hostextinfo.cfg
#cfg_file=/usr/local/nagios/etc/serviceextinfo.cfg
```

Now I've uncommented or defined these configuration files to Nagios, I need to create empty files to hold our configuration objects. I do this using the `touch` command, as you can see here:

```
puppy# cd /usr/local/nagios/etc
puppy# touch contacts.cfg contactgroups.cfg hosts.cfg hostgroups.cfg➥
services.cfg servicegroups.cfg timeperiods.cfg
puppy# ls *.cfg
bigger.cfg         contacts.cfg    misccommands.cfg   services.cfg
cgi.cfg            hostgroups.cfg  nagios.cfg         timeperiods.cfg
checkcommands.cfg  hosts.cfg       resource.cfg
contactgroups.cfg  minimal.cfg     servicegroups.cfg
```

The `touch` command will create a series of empty files that I can fill with my configuration objects.

■ **Caution** Having the empty files will still not allow the Nagios server to start. It will detect the empty files and determine that no configuration objects have been defined and refuse to start. I'll have to define some objects before I can start the Nagios server.

There is another way to define configuration files to the Nagios server: you can tell Nagios to include all files with a `.cfg` extension contained in a specified directory using the `cfg_dir` directive. Example 2-3 shows how to use the `cfg_dir` directive.

Example 2-3. *The cfg_dir Directive*

```
cfg_dir=/usr/local/nagios/etc/192.168.0
cfg_dir=/usr/local/nagios/etc/192.168.1
cfg_dir=/usr/local/nagios/etc/192.168.2
cfg_dir=/usr/local/nagios/etc/192.168.3
```

In Example 2-3 all files in the directories listed would be parsed by the Nagios server and included in the configuration. This is useful if you have a large number of configuration files and you want to further organize them into categories. For instance, you would create a directory to hold all hosts in a particular subnet while further dividing your objects into types inside configuration files in that directory. Example 2-4 shows a model like this.

Example 2-4. *Categorized Configuration Files*

```
puppy# ls -l 192.168.0
total 16
-rw-r--r--  1 nagios nagios 0 May 12 20:46 firewalls.cfg
-rw-r--r--  1 nagios nagios 0 May 12 20:46 routers.cfg
-rw-r--r--  1 nagios nagios 0 May 12 20:46 servers.cfg
-rw-r--r--  1 nagios nagios 0 May 12 20:46 switches.cfg
```

The 192.168.0 directory contains a series of configuration files that contain the definitions of host objects of different types—for example, `firewalls.cfg` could hold the object definitions for all the firewalls in the 192.168.0 subnet.

Finally, all configuration files should be owned by the user and group that is running the Nagios server process. In our case, the user and group are both `nagios`. You should change their ownership prior to attempting to start the Nagios server. The command on the following line changes the ownership of all files ending in `.cfg` in the `/usr/local/nagios/etc` directory and all directories below:

```
puppy# chown -R nagios:nagios /usr/local/nagios/etc/*.cfg
```

Defining Nagios Configuration Objects

Inside each configuration file that you specify to Nagios, you define the objects required for your Nagios configuration. Each object definition is created by combining a series of directives. The directives represent the attributes and settings of the object being defined. These directives can either be unique to the type of object being defined—for example, a host object has a directive that defines its address, or generic and applicable to a number of object definitions.

Some directives in an object definition are mandatory. For instance, for a host object definition you must provide a hostname and an address. If you do not provide these mandatory directives in the object definition, the definition will not be valid. An invalid definition will cause the Nagios server to fail when you attempt to start it. Other directives are optional, and excluding them will not impact the validity of the object definition. For each directive I examine, I'll specify whether it is mandatory or optional to a particular object definition.

The directives are contained within an overarching directive called the `define` directive. Example 2-5 shows the construction of a `define` directive.

Example 2-5. *The define Directive*

```
define    object_type{
           directive1 setting
           directive2 setting
           directive3 setting
           }
```

As you can see from Example 2-5, the `define` directive consists of the directive name followed by the name of the type of object you wish to define (see Table 2-1 for the available object names). You would replace `object_type` in Example 2-5 with the name of the object type you wish to define—such as `host` for host objects, as you can see in the next few lines:

```
define    host{
           directive1 setting
           }
```

The name of the object type is followed directly (with no space in between) by an opening { bracket. On the following lines, with one directive to a line, are the directives describing that object. These directives consist of the directive name and, separated by spaces or tabs, the directive value. Finally, the `define` directive is closed with a } bracket.

You can also place comments in your object definitions by prefixing them with a ; symbol, as you can see here:

```
define    object_type{
           directive1 setting ; this is a comment
           }
```

Defining Your First Host

In this section I'm going to look at host object definitions and also introduce quite a few of the topics you'll need to understand about Nagios and how it operates. I'll look at each of these topics when introducing the directives related to these topics. Many of these topics will also be used with other object definitions, such as services. Indeed, many of the same directives used to define hosts are also used to define services and other object types.

Host objects represent the physical devices on your network like servers, routers, switches, or other pieces of infrastructure. Another way of looking at host objects is that they are items or assets on your network that can be connected to via some sort of address—for example, an IP address or a media access control (MAC) address. Each host object definition consists of a large number of potential directives, some mandatory and some optional.[1] In Example 2-6 I've provided a typical host definition that contains all of the mandatory directives required for a host definition. You must include at least these directives in your host definition or the definition will be considered invalid.

Example 2-6. *Host Object Definition*

```
define host{
            host_name                   kitten.yourdomain.com
            alias                       Primary Sydney Server
            address                     192.168.0.1
            check_period                24x7
            max_check_attempts          1
```

1. You can see a full list of the host object directives at `http://nagios.sourceforge.net/docs/2_0/xodtemplate.html#host`.

```
contact_groups              network_team,field_support
notification_interval       30
notification_period         24x7
notification_options        d,u,r
}
```

Let's look at Example 2-6 and see how it works. First, we've defined a host called `kitten.yourdomain.com` with an address of 192.168.0.1 that we've described as our `Primary Sydney Server`. We've defined that it should be checked during a time period called `24x7`. If something goes wrong, we want to send notifications during a time period called `24x7` to the `network_team` and the `field_support` team. These notifications should be sent every `30` minutes. In this section, I'll look at all the directives that define what I've just described. I'll also look at the further directives available for your host definitions. To do this, I'll step through each major function provided by the directives, starting with some basic directives and moving through checking your host and configuring host notifications and then the remaining host definition directives.

Defining the Hostname and Address

Let's start with some of the basic directives of a host definition, as shown in Table 2-2.

Table 2-2. *Names and Address Directives*

Directive	Description	Macro	Mandatory?
`host_name`	Name of the host	`$HOSTNAME$`	Yes
`alias`	Longer description of the host	`$HOSTALIAS$`	Yes
`address`	Address of the host	`$HOSTADDRESS$`	No (but strongly recommended)

Table 2-2 contains four columns. The first two columns indicate the directive name and description. The third and fourth columns indicate if the directive's value can be used as a macro and whether it is mandatory to the object definition. An object definition must have all the required mandatory directives included in the definition to be valid. An invalid object definition will result in Nagios failing to start. I'll discuss macros in a bit and in the "Defining Commands" section later in this chapter.

The first directive you need to define is the `host_name` directive. This directive is the name of the host being defined. Nagios calls this the *short name* of the host. It is used in other object definitions to reference the host object. For example, when adding host objects to a host group, you would refer to each host object by the name defined in the `host_name` directive. It is preferable that you use the actual hostname of the host being defined in this field. This is because if you don't specify an address for the host, Nagios will attempt to use Domain Name Service (DNS) to resolve the contents of the `host_name` directive to find the address of the host. If DNS is unavailable or you suffer a DNS outage, Nagios will be unable to resolve the address of the host and hence unable to monitor the host or any of its services.

The `host_name` directive can also be used as a macro. Macros are one of the most useful features in Nagios. Macros are generally used in the commands you use to check hosts and services. Instead of having to specify a particular hostname or address in each command, you specify a macro, which is replaced at the time the command is run with the value required. This allows you to define generic commands that can be used to monitor multiple hosts and

services instead of having to define a command to monitor each host or service. I'll discuss macros more in the "Defining Commands" section.

You can identify macros because they are prefixed and suffixed with the $ symbol—for example, the value of the `host_name` directive is contained in a macro named `$HOSTNAME$`.

The `host_name` directive is linked to the next directive, `alias`, which defines a longer name or description for the host object. This allows you to provide a more descriptive name for the host object. The `alias` directive is also represented by a macro, `$HOSTALIAS$`.

■ **Note** If you do not specify a value for the `alias` directive, it defaults to the value of the `host_name` directive. This includes the value of the `$HOSTALIAS$` macro.

The next directive, `address`, is used to define the address of the host. In most cases this will be an IP address, but it can be any type of address that can be used to check the status of the host. For instance, you could use radio frequency identification (RFID) addresses, a supervisory control and data acquisition (SCADA) address, or a MAC address if you had a command that could check the status of hosts using these types of addresses.

As I discussed earlier, if you don't specify an address for the host and leave this directive blank or exclude it, Nagios will try to use the contents of the `host_name` directive to determine the address of the host. It will assume the connection is a TCP/IP connection and attempt to resolve the value in the `host_name` directive using DNS. If it is not a TCP/IP connection, or you do not have DNS resolution on your Nagios server, then you won't be able to check the host's status. This is also true if your DNS name resolution fails. Nagios will then also be unable to check the host. I recommend you always specify an address for your host using this directive.

■ **Tip** The host address also has a macro associated with it, `$HOSTADDRESS$`. This is frequently used in commands to specify the address of your host.

Parents, Host Groups, and Contact Groups

The next directives relate to defining other hosts that your host might be dependent on, any groups of hosts it belongs to, and who to notify in the event that an error is detected on the host. I've displayed the relevant directives in Table 2-3.

Table 2-3. *Names and Address Directives*

Directive	Description	Mandatory?
`parents`	Specifies the parent hosts of this host	No
`hostgroups`	Specifies any host groups this host belongs to	No
`contact_groups`	Contact groups that should be notified for this host	Yes

The first directive in Table 2-3 is the `parents` directive. The `parents` directive lets you specify other assets that your host is dependent on to function. This could include a switch, router, or firewall that your host relied on for connectivity. If the parent host is unavailable, all child hosts will also be unavailable. Any parent hosts you define using the `parents` directive also need to be defined to Nagios using host objects.

You specify any parent hosts in your `parents` directive using the value of the `host_name` directive of the parent host or hosts. If one of these intervening hosts is not available when checked by Nagios, all hosts for which that host is a parent will be changed to a state of `UNREACHABLE` (see the "Host States" sidebar). You can specify multiple parent hosts in the directive by separating them with commas:

```
parents        melb_router,melb_switch1,melb_switch2
```

The next directive is the `hostgroups` directive. This directive lets you specify the host group or groups that the host you are defining belongs to. Host groups are a means of grouping together like hosts—for example, by type or location. You can specify that the host belongs to multiple host groups by specifying the list of host groups separated by commas:

```
hostgroups          servers,aust_assets
```

The `contact_groups` directive is a list of the contact groups that should be notified when there is a problem with this host. Contact groups are groupings of the people, or contacts, who may need to be notified when an event or state change occurs. For instance, you could group together all your Network Management staff or all the IT staff in a particular site into contact groups.

You can specify multiple contact groups here by separating each group with commas. You can see that in Example 2-6 where I've specified that the contact groups `network_team` and `field_support` should be notified in the event of a problem with the `kitten` host.

■ **Note** All of your contacts should belong to at least one contact group. I'll discuss host and contact groups further in the "Grouping Objects" section.

Checking the Host

Now you have defined the basics of your host, you need to define the directives that relate to checking the host, including the command you want to use to check the host and when and how often it should be checked. You can see these directives in Table 2-4.

Table 2-4. *Host Check Directives*

Directive	Description	Mandatory?
check_command	Command used to check the state of the host	No
check_period	Specifies when to perform checks of the host	Yes
max_check_attempts	The number of check attempts to make before generating a notification	Yes
check_interval	Period in between checks of the host	No
active_checks_enabled	Specifies that active checks are enabled for this object	No
passive_checks_enabled	Specifies that passive checks are enabled for this object	No
check_freshness	Enables freshness checks for this object	No
freshness_threshold	Specifies the freshness threshold for this object	No

The first of these directives is the check_command directive, which tells Nagios what command to use to check the state of the host object. Most host checks are performed to confirm that the host is active and responding to checks; usually this is done with a ping.

If you do not include the check_command directive in your host object definition or leave it blank, Nagios will not check the state of the host. Nagios will assume the host is always up. This is useful for hosts that are often off, such as printers or photocopiers. This prevents you from receiving false positives for devices that have been deliberately turned off.

■ **Tip** The check_command is available as a macro called $HOSTCHECKCOMMAND$.

The check_command directive also has a related directive, host_check_timeout, in the nagios.cfg configuration file. The host_check_timeout directive controls how long Nagios will wait for a response from the check command before timing out. By default, it set to 60 seconds:

```
host_check_timeout=60
```

If the timeout is reached, Nagios will assume the host is unavailable, put it in a DOWN state, and log an error indicating that the command timed out. The timeout is mostly designed to manage runaway checks that have not exited correctly. If necessary you should adjust this to reflect the network latency in your environment. In most cases the default will be fine.

■ **Note** I'll look at commands in the "Defining Commands" section later in this chapter.

The next directive is check_period. This directive tells Nagios when it should check the host. For example, you may wish to check some hosts all of the time and others only during set periods. The check_period directive is linked to another object type, the timeperiod object. The timeperiod object defines periods of time—for example, business hours Monday to Friday

or 24 hours every day of the week. Nagios uses these defined time periods to specify when to monitor or take action on events it detects. In Example 2-6 I've specified a time period called `24x7`. I'll demonstrate how to define this period and others later in the "Defining Time Periods" section. The value of the `check_period` directive should be the value of the `timeperiod_name` directive from the time period object definition. You can specify more than one time period by separating them with commas.

The `max_check_attempts` directive tells Nagios how many times to retry checks on a host when a check returns a non-`OK` state. Until Nagios has retried the checks the requisite number of times it will not generate a notification. For example, Nagios checks the status of the `kitten` host and determines that it is not responding. Rather than immediately change the status of the host, Nagios rechecks the host the number of times specified in the `max_check_attempts` directive. If at the end of this rechecking the host is still not responding, then Nagios will change its state and, if configured to, generate a notification.

The setting of the directive is a numeric value. A value of `1` means that if one check fails Nagios will generate a notification immediately. A value of `2` will result in Nagios checking the host twice and if after the second check the result is still not `OK`, then it will generate a notification. You can specify as many extra checks as you like here. This directive is mandatory and you should always specify a value of at least `1` even if you don't want to check the status of the host. As discussed earlier, if you want to disable host checking you can leave out the `check_command` directive or leave it blank.

HOST STATES

When Nagios checks a host or service, that check returns the state of that host or service. There are different states for hosts and services. For hosts there are three possible states: `OK` and two error states, `DOWN` and `UNREACHABLE`. The `OK` state indicates that the host is up and available. The `DOWN` state indicates that the host is unavailable. For example, if you are using a ping to ascertain the status of the host, this means the ping was not received and the host itself is unavailable or not able to be contacted by the Nagios server. The `UNREACHABLE` state indicates that for some reason the command was unable to reach the host—for example, the network is down. This doesn't always mean your host is down but generally that the network or some network component between your Nagios server and your host is unavailable. This is also the state used when you have defined dependencies or parents for your hosts. When a host or hosts your host is dependent on has failed, your host is usually marked as `UNREACHABLE`.

State types add another layer of complexity to your states. State types apply to both host and service states. There are two state types: soft states and hard states. A soft state can almost be described as a "pending" state. Let's look at a brief example. A host object has the `max_check_attempts` directive set to `3` check attempts. This means that if Nagios checks the host and it returns a state of `DOWN`, it will retry a check of the host three times. In the period during these checks, Nagios considers the host to be in a soft `DOWN` state. This means Nagios has not fully determined that it is actually `DOWN` and is waiting until all the required checks have been performed before finalizing the state of the host. When Nagios has completed all these retry checks and if the host still returns a `DOWN` state, then this state is now considered a hard `DOWN` state. This indicates that Nagios now believes that the host is definitely `DOWN` and that a notification could be generated (if Nagios is configured to do that). If, however, the host recovers from the `DOWN` state and returns to an `OK` state during this period of checks, Nagios calls this a soft recovery. The host never reaches the hard state and a notification is not generated. This soft and hard state model helps reduce the number of potential false alarms.