Extracted from:

Deploying Rails

Automate, Deploy, Scale, Maintain, and Sleep at Night

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Deploying Rails

Automate, Deploy, Scale, Maintain, and Sleep at Night

Anthony Burns
and Tom Copeland

Edited by Brian P. Hogan

6.6 Monitoring Applications

The monitoring techniques we've seen have all been useful with any application; check_disk is helpful for a Ruby on Rails application, a Java application, or a static website, and check_passenger is useful for any Rails application. But Nagios can also monitor an application's specific data thresholds. Let's see how that's done.

We've seen how Nagios uses plugins to bring in new functionality, and we've written our own plugin to monitor Passenger memory usage. Now we'll write another plugin to check MassiveApp's activity. Actually, most of the logic will be in MassiveApp; we'll write just enough of a plugin to connect to MassiveApp and report a result.

For the specifics of this check, consider MassiveApp's daily growth rate in terms of accounts. In any twenty-four hours, we get around a dozen new accounts. If we get many more than that, we want to get an alert so we can think about firing up more servers.

We could do this check in a few different ways. We could query the MySQL database directly, but although that would avoid the performance impact of loading up the Rails framework, it would mean we couldn't use our ActiveRecord models with their handy scopes and such. We could use HTTP to hit a controller action, but then we'd want to ensure that the action could be accessed only by Nagios. So, we'll keep it simple by using a Rake task. First we'll declare the task; we can put this in lib/tasks/monitor.rake. We're namespacing the task inside nagios; this keeps all our Nagios-related tasks in one place.

monitoring/task_only/lib/tasks/monitor.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
  end
end
```

Next let's count the number of "recently" created accounts; in this case, "recently" means "in the past twenty-four hours." We can do this with a straightforward ActiveRecord query.

monitoring/task_and_query/lib/tasks/monitor.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
    recent = Account.where("created_at > ?", 1.day.ago).count
  end
```

```
end
```

In this plugin, we'll ask for a warning if we get more than fifty new accounts in a day, and we'll consider it critical if we get more than ninety in a day.

monitoring/lib/tasks/monitor.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
    recent = Account.where("created_at > ?", 1.day.ago).count
    msg, exit_code = if recent > 90
      ["CRITICAL", 2]
    elsif recent > 50
      ["WARNING", 1]
    else
      ["OK", 0]
    end
    puts "ACCOUNTS #{msg} - #{recent} accounts created in the past day"
    exit exit_code
  end
end
```

We have the Rake task in MassiveApp's codebase now; next up, we need Nagios to be able to run it. We can do this with a simple Bash script. We'll name this script check_recent_accounts and put our nagios Puppet module in modules/nagios/files/plugins/ alongside our check_passenger plugin. That script needs to run our Rake task using the --silent flag to prevent the usual "(in /path/to/the/app)" Rake output message since that would confuse Nagios. It also needs to relay the exit code from the Rake task on to Nagios. We can do that using the Bash special parameter $?, which holds the exit code of the last command executed.

monitoring/modules/nagios/files/check_recent_accounts

```
#!/bin/bash
cd /var/massiveapp/current/
RAILS_ENV=production /usr/bin/rake --silent nagios:accounts
exit $?
```

Switching back into Puppet mind-set, we'll add another file resource to our nagios::client class that will move our script into place.

monitoring/check_recent_accounts.pp

```
"/usr/lib/nagios/plugins/check_recent_accounts":
   source  => "puppet:///modules/nagios/plugins/check_recent_accounts",
   owner   => nagios,
   group   => nagios,
   mode    => 755;
```

And as with check_passenger, we'll want to ensure the package is installed before attempting to copy our script into place. (See Figure 4, *Ensure the package is installed*, on page 8)

We'll also need to add the new command to nrpe.cfg.

```
command[check_recent_accounts]=/usr/lib/nagios/plugins/check_recent_accounts
```

Let's run Puppet to get the script and the new nrpe.cfg in place. We can get into the console and add a few test accounts just to get data to work with.

```
app $ ./script/rails console production
Loading production environment (Rails 3.2.2)
ruby-1.9.3-p194 :001 >\
 70.times {|i| Account.create!(:email => "test#{i}@example.com") }; nil
```

Now we can execute a trial run of this plugin in the same way that we've exercised other plugins; we'll just run it from the shell.

```
app $ /usr/lib/nagios/plugins/check_recent_accounts
ACCOUNTS WARNING - 70 accounts created in the past day
```

We need to tell Nagios about our new check, so we'll add it to commands.cfg.

monitoring/commands.cfg

```
define command {
  command_name check_recent_accounts
  command_line /usr/lib/nagios/plugins/check_recent_accounts
}
```

And we'll add this to our app checks:

monitoring/nagios-service-definitions

```
define service {
  use generic-service
  service_description Recent Accounts
  host_name app
  check_command check_nrpe_1arg!check_recent_accounts
}
```

Another Puppet run, and everything is in place; now Nagios will let us know if (or when) we get a mad rush of new users.

This is the sort of check that needs to be run only once for MassiveApp. That is, when MassiveApp grows to encompass a few servers, we won't run this on each server as we'd do with check_ssh and check_passenger. Instead, we'd designate one host to run this check and to alert us if the thresholds were exceeded.

```
package {
  ["nagios-nrpe-server","nagios-plugins"]:
    ensure => present,
    before => [File["/etc/nagios/nrpe.cfg"],\
    File["/usr/lib/nagios/plugins/check_recent_accounts"],\
    File["/usr/lib/nagios/plugins/check_passenger"]]
}
```

Figure 4—Ensure the package is installed

6.7 Where to Go Next

This whirlwind tour of Nagios has hit on the high points, but there are some interesting areas for future investigation. The ones we think are important are learning more about notifications and escalations, further exploring the standard Nagios interface, and selecting the appropriate Nagios tools and services to meet the needs of your system.

Notifications and Escalations

Nagios allows for a variety of notification mechanisms and strategies; SMS and email are built in, but plugins are available for notifying via IRC, Twitter, Campfire, and more. We can also set up per-person notification schedules so that a particular person doesn't receive notifications on their regular day off or receives only critical notifications for critical hosts on that day. Using contacts, hosts, and hostgroups to tune notifications can prevent everyone from seeing and eventually ignoring too many notifications.

If a notification is sent and not handled within a certain time period, Nagios can then escalate that notification to another group. An initial notification might go to a help-desk support group, and if the problem persists, another notification can be sent to a sysadmin group. Nagios has serviceescalation and hostescalation objects to help manage the escalation schedules and strategies.

One system failing can cause others to fail and result in an avalanche of notifications. We can manage this by setting up dependencies so that if one system drops offline, Nagios will suppress the messaging for dependent services. Nagios allows for both host and service dependencies to further throttle messaging as needed. And in a pinch, we can even disable messaging systemwide if something has gone terribly wrong.

Exploring the Standard Nagios Interface

We touched on a few areas of the built-in Nagios interface, but there's a lot more to see there. It includes pages that provide a topological map of the monitored hosts, it has an interface that enables access via a Wireless Markup Language client, and it even includes a (somewhat gratuitous) Virtual Reality Markup Language view. More practically, there are pages showing event histograms, a history of all notifications, histories of all log entries, and the ability to view event trends over time. It also allows for planned downtime windows to be created that will suppress notifications during that time. So, although the Nagios interface sometimes appears a bit dated, there's a lot of functionality there.

For those who want enhancements to the standard interface, there are several tools (NagVis, NDOUtils, exfoliation, NagiosGraph, and more) that can supply different views into the data that Nagios collects. Several of these enhancements show additional data; for example, NagiosGraph can display a timeline of the performance reported by each Nagios check.

Nagios Ecosystem

Nagios has a rich ecosystem of options to meet various needs. For example, connecting a central host to monitored hosts may not be the right solution for everyone. Nagios also provides an alternative way, *passive checks*, that uses the Nagios Service Check Acceptor to report checks to the central monitoring server. With passive checks, the Nagios server doesn't reach out to agents. Instead, other processes push check results into a directory where the server picks them up. This can work well for large installations and those where the server doesn't have access to the hosts that need to be monitored.

We've looked at a few plugins and written a simple one. But some of the plugins open doors to entire new areas of functionality; for example, the `check_snmp` plugin enables Simple Network Management Protocol integration. The Nagios Exchange[3] includes hundreds of ready-made plugins; it's worth reviewing the recent releases section occasionally to see what people are writing for Nagios.

3. http://exchange.nagios.org/