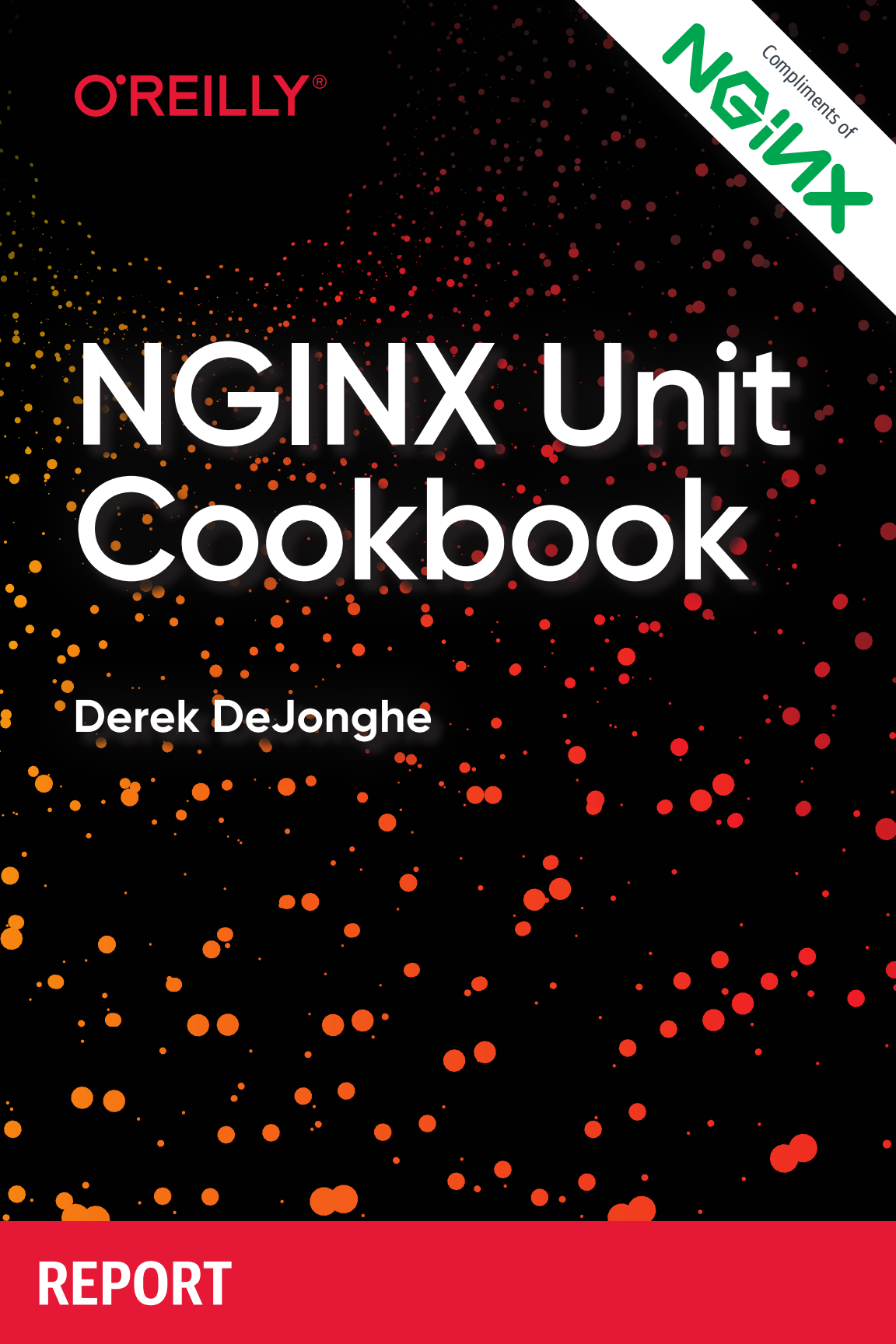
O'REILLY®
Compliments of
NGINX
NGINX Unit Cookbook
Derek DeJonghe
REPORT

NGINX Unit Cookbook

Derek DeJonghe

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

NGINX Unit Cookbook
by Derek DeJonghe

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Mary Treseler
Developmental Editors: Nikki McDonald and Eleanor Bru
Production Editor: Nan Barber
Copyeditor: Arthur Johnson
Proofreader: Nan Barber
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2019: First Edition

Revision History for the First Edition
2019-06-11: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492054306* for release details.

This work is part of a collaboration between O'Reilly and NGINX. See our statement of editorial independence.

978-1-492-05428-3

LSI

Table of Contents

CHAPTER 1

Unit Introduction and Features

Introduction

This chapter will introduce you to NGINX Unit in a traditional book format before switching to the O'Reilly Cookbook format in Chapter 2. Throughout this chapter you will learn about what makes Unit different from other middleware application servers. Before learning the how, you'll learn the why, with a brief history of the problem Unit aims to solve. From that understanding, the architecture of NGINX Unit will be introduced, followed by the language support, and finally the API that drives the configuration.

Application Landscape and Unit Project History

The landscape of web applications has changed. In the past, applications were written from the ground up to serve specific needs, and upgrades were seldom issued compared to the present day. Today, applications are released frequently, in piecemeal fashion, and portions are completely rewritten over time. As teams and web application offerings grow, the likelihood of the logic being diverse in both language and code base grows as well.

As web applications diversify through microservices, languages, and language versions, so does the operational complexity of managing middleware, where middleware is defined as the application server that receives requests and ushers them to the application code.

Installing, configuring, tuning, and maintaining multiple types of middleware servers for different types of application languages and versions requires a lot of work, expertise, and time and affects the bottom line.

The team at NGINX Inc. has observed this change in the application landscape and has worked to develop a solution from scratch, one that is built for the new age of computing. This solution, NGINX Unit, aims to reduce operational complexity by providing a single middleware server that is able to run multiple applications of different languages and versions and to update on the fly without dropping a connection.

Dynamic Application Server

NGINX Unit is a dynamic application server, which means that it can be dynamically reconfigured during runtime without dropping requests. The architecture of Unit is such that request handling is broken into layers. These layers comprise a control process, a router process, and some application processes.

Each application served by Unit is run by an isolated process or set of processes. The router process receives incoming connections and asynchronously queues them for the destined application. The control process manages the configuration of the application and routing processes. The administrator, or operational automation, interacts with the control process through an application programming interface (API). The control process is able to reconfigure routing and application processes on the fly.

Polyglotism

Polyglotism is the ability to speak multiple languages. Prior to NGINX Unit, a few polyglot middleware services have served the web well—for example, the Common Gateway Interface (CGI) supports languages such as PHP, Perl, and Python; the Web Server Gateway Interface (WSGI) supports Perl, Python, and Ruby. Unit provides a single middleware server to run both compiled and scripting languages—including the aforementioned languages as well as Node.js, Go, and Java—through a unified configuration.

With NGINX Unit, teams are able to code in the application language that makes the most sense for the service they're providing to

the end user. This technology reduces the difficulty of running complex systems to enable business value from all aspects.

API-Driven Configuration and Server Management

The NGINX Unit control process is advertised through an API. The API can be configured to be served through a Unix or TCP socket. These two options allow the API to be tightly controlled but also enable remote configuration. This API follows RESTful paths, methods, and JSON bodies, as per industry standard.

The control process is able to start and stop application processes and to reconfigure only necessary portions of the routing process's memory. This ability to start applications and configure traffic routing accordingly is the core of the dynamic reconfiguration. These paradigms enable native integration with operational workflows found in DevOpsian organizations.

CHAPTER 2

Installation

Introduction

The first step toward using NGINX Unit is installing it. NGINX Unit can be installed on a wide variety of systems. This chapter will detail how to install Unit on the major Linux distributions such as Debian, Ubuntu, Red Hat, and CentOS through NGINX package repositories. Other installation methods, such as compiling from source and using third-party repositories, are also included to enable success with NGINX Unit on virtually any Linux-based platform.

Red Hat–Based Systems (.rpm)

Problem

You need to install NGINX Unit on Red Hat or CentOS.

Solution

Create a file named */etc/yum.repos.d/unit.repo* that contains the following contents:

```
[unit]
name=unit repo
baseurl=https://packages.nginx.org/unit/OS/$releasever/$basearch/
gpgcheck=0
enabled=1
```

Alter the file, replacing `OS` at the end of the URL with `rhel` or `centos`, depending on your distribution.

Install the Unit base package:

```
sudo yum install unit
```

Install additional modules that you may want to use with Unit:

```
sudo yum install unit-php unit-python unit-go unit-perl \
      unit-devel unit-jsc-common unit-jsc8 unit-jsc11
```

Discussion

The file you just created for this solution instructs the `yum` package management system to utilize the Official NGINX Unit package repository. The command that follows installs Unit from the Official repository, as well as the Unit modules needed for each application language you may want to run.

Additional Resources

System Requirements
CentOS Package Documentation
RHEL Package Documentation

Debian-Based Systems (.deb)

Problem

You need to install NGINX Unit on a Debian or Ubuntu machine.

Solution

Ensure that the Advanced Package Tool (APT) system is able to use HTTPS repositories:

```
sudo apt-get install apt-transport-https
```

Create a file named */etc/apt/sources.list.d/unit.list* that contains the following contents:

```
deb https://packages.nginx.org/unit/OS/ CODENAME unit
deb-src https://packages.nginx.org/unit/OS/ CODENAME unit
```

Alter the file, replacing `OS` at the end of the URL with `ubuntu` or `debian`, depending on your distribution. Replace `CODENAME` with the

code name of your system. If you don't know the code name the following command will output the value you need:

```
lsb_release -c
Codename:       xenial   # Example
```

Run the following commands to install the NGINX signing key and install Unit:

```
wget http://nginx.org/keys/nginx_signing.key
sudo apt-key add nginx_signing.key
sudo apt-get update
sudo apt-get install unit
```

A version of the language needs to be specified for certain Unit modules. At the time of this writing, not all versions of all languages are supported across all versions of the OS. You can search for module packages available from the repository for your operating system by using the following command:

```
apt-cache search unit- | grep NGINX
```

Install additional modules that you may want to use with Unit. The following packages are available on all Debian-based systems:

```
sudo apt-get install unit-php unit-python2.7 unit-perl \
      unit-ruby unit-dev unit-jsc-common unit-jsc8
```

Discussion

The file you just created instructs the `apt` package management system to utilize the Official NGINX Unit package repository. The commands that follow download the NGINX GPG package signing key and import it into `apt`. Providing the APT system with the signing key enables it to validate packages from the repository. The `apt-get update` command instructs the APT system to refresh its package listings from its known repositories. After the package list is refreshed, you can install Unit and any necessary packages from the Official NGINX repository. For Python 3 and Golang, not all minor versions are supported on all systems. The search command demonstrated previously can assist in finding which language versions are available for your system.

Additional Resources

System Requirements
Debian Package Documentation
Ubuntu Package Documentation

Third-Party Repositories

Problem

You want to run NGINX Unit on a system for which NGINX Inc. does not have prebuilt packages, and you do not want to build from source.

Solution

Install from a third-party repository. These named repositories are maintained by the community; NGINX has no control or responsibility over these resources.

Third-Party Repositories

These third party repositories are maintained by the community. NGINX Inc. is not responsible for them or for what gets installed when using them.

Alpine Linux:

```
sudo apk update
sudo apk upgrade
sudo apk add unit
sudo apk add unit-openrc unit-perl unit-php7 unit-python3 unit-ruby
```

Arch Linux:

```
sudo pacman -S git
git clone
git clone https://aur.archlinux.org/nginx-unit.git
cd nginx-unit
makepkg -si
```

FreeBSD:

```
sudo pkg install -y unit
```

Gentoo:

```
sudo emerge --sync
sudo emerge www-servers/nginx-unit
```

Remi's RPM repository hosts the latest version of PHP for RHEL and its derivatives such as CentOS and Fedora:

```
sudo yum install --enablerepo=remi unit \
      php54-unit-php php55-unit-php php56-unit-php \
      php70-unit-php php71-unit-php php72-unit-php php73-unit-php
```

Unit's Node.js package is called unit-http. It uses Unit's libunit library; your Node.js applications require the package to run in Unit:

```
sudo npm install -g --unsafe-perm unit-http
```

Discussion

This section has detailed the usage of a number of third-party repositories maintained by the community. It is possible to utilize this information to quickly install prebuilt Unit and Unit module packages on systems that NGINX Inc. does not yet maintain a repository for. Also, the Remi repository contains specific older PHP versions that may be useful to some readers.

Additional Resources

System Requirements
Community Repositories Install Documentation

Installing from Source

Problem

You need to install Unit from source.

Solution

You will have to install the packages needed to compile from source. The following includes all the development packages for all supported languages; skip the packages that you are not going to use.

For Debian and Ubuntu:

```
sudo apt-get install build-essential
sudo apt-get install golang
```

```
sudo curl -sL \
      https://deb.nodesource.com/setup_<Node.js version>.x \
      | bash -; apt-get install nodejs
sudo apt-get install php-dev libphp-embed
sudo apt-get install libperl-dev
sudo apt-get install python-dev
sudo apt-get install ruby-dev
sudo apt-get install libssl-dev
```

For Amazon Linux, CentOS, RHEL, and Fedora:

```
sudo yum install gcc make unzip
sudo yum install golang
sudo curl -sL \
      https://rpm.nodesource.com/setup_<Node.js version>.x \
      | bash -; yum install nodejs
sudo yum install php-devel php-embedded
sudo yum install perl-devel perl-libs
sudo yum install python-devel
sudo yum install ruby-devel
sudo yum install openssl-devel
```

Clone or download the source code from *https://github.com/nginx/unit*. If you choose to download, you'll need to unzip the package that is downloaded. Once the source is cloned or unpacked, move into the base of the project. The next example follows the download path:

```
curl -O https://codeload.github.com/nginx/unit/zip/master
unzip master
cd unit-master/
```

Next, use the `configure` script to prepare the source code for installing on your system. Run `./configure --help` to fully understand the flags available. In the following example, the `--prefix` option is used to specify the installation directory. Each supported language has an associated module that also needs to be built. Run the `configure` script with each application type you need to build a module for:

```
./configure --prefix=/opt/unit/
./configure go
./configure perl
./configure php
./configure python
./configure ruby
```

Next, use the `make` command to run the *Makefile* created by the `configure` script and install the software. You will need to run the `make` command for each language. Depending on the location and

ownership of the `--prefix` flag specified by the `configure` command, you may need to run the last command with elevated privileges:

```
make
make perl
make php
make python
make ruby
sudo make go-install
sudo make node-install
sudo make install
```

NGINX Unit is now installed. Validate the installation by getting the help options from the binary:

```
sudo /opt/unit/sbin/unitd -h
```

Discussion

The preceding steps will build and install NGINX Unit from source. A number of configuration flags can be used to modify the build and installation. Unit is ready to use.

Additional Resources

System Requirements
Source Installation Documentation

CHAPTER 3

Configuration

Introduction

There are three main configuration objects used by NGINX Unit. All are defined with JSON. The application object defines characteristics of the application being run by Unit, such as the language, the process controls, and the location on the filesystem. The listener object defines the Unit configuration that directs incoming requests on a defined IP address and port to a specified application. The route objects provide internal routing capabilities. This chapter will build a foundational understanding of these objects.

Application Object

Problem

You need to understand the application object for a fundamental understanding of NGINX Unit.

Solution

Define an application object that describes an application on the system. Each application type has different attributes and options that can be applied. The following is a basic example of a PHP application object:

```
{
    "applications": {
        "my-app": {
```

```
            "type": "php",
            "processes": 2,
            "root": "/var/www/app/",
            "index": "index.php",
            "user": "app_user",
            "group": "app_group"
        }
    }
}
```

Discussion

Every application deployed on NGINX Unit is defined by an application object. The application object, defined in JSON, specifies the attributes of the application. Each application type has its own required and optional attributes. A number of different application attributes control Unit process management and limitation. The `type` attribute is the only process management attribute that is required for an application; it defines the application language, such as PHP, Python, Golang, Ruby, or Perl. Other attributes include limits on child process count, request time, user, group, environment variables, and working directory.

In the example, some of the attributes that can be applied to a PHP process are used, such as `root` and `index`. The application-specific attributes are focused on the entry point of the application, such as the directory of the project, or main executable file.

You will learn how to apply application objects to the Unit configuration in the section "Applying Configuration" on page 20.

Additional Resources

Applications Object

Listener Object

Problem

You need to understand the NGINX Unit listener object in order for your application to listen for requests.

Solution

Define a listener object to instruct Unit to listen for incoming requests on a provided IP and port:

```
{
    "listeners": {
        "*:8080": {
            "pass": "applications/my-app"
        }
    }
}
```

Discussion

To instruct NGINX Unit to listen on an IP and port, a listener object must be defined. The listener object defines the application to which Unit will direct incoming requests. The listener object is the value, specified to a key that defines the IP and port. In the example, the `*` is used for the IP address, thus instructing Unit to listen on all IP addresses. The listener object has two attributes: `pass` and optionally `tls`. The `pass` attribute takes a string value that specifies the application or route to which requests will be directed. The example sends requests directly to an application named `my-app`. The `pass` attribute replaced the `application` attribute of the listener object in version `1.8.0`.

You will learn how to apply listener objects to the Unit configuration in the section "Applying Configuration" on page 20.

Additional Resources

Listeners Object

Route Object

Problem

You want to understand the NGINX Unit route objects to enable internal routing between listeners and applications.

Solution

The `routes` attribute of the Unit configuration can be configured as an array of route objects, or an object of named route arrays. The

difference of configuration alters the usage of the listener object `pass` attribute.

When an array of route objects is used as the value of the `routes` attribute, the value provided to the `pass` attribute is simply `routes`, as in the following example:

```
{
    "listeners": {
        "*:8080": {
            "pass": "routes"
        }
    },
    "routes": [
        {
            "match": {
                "host": "blog.example.com"
            },
            "action": {
                "pass": "applications/blog"
            }
        },
        {
            "action": {
                "pass": "applications/my_app"
            }
        }
    ]
}
```

When an object of named route arrays is used as the value of the `routes` attribute, the value provided to the `pass` attribute must be `routes/` followed by the named route, as in the following example:

```
{
    "listeners": {
        "*:8080": {
            "pass": "routes/main"
        }
    },
    "routes": {
        "main": [
            {
                "match": {
                    "host": [ "example.com", "www.example.com" ]
                },
                "action": {
                    "pass": "applications/website"
                }
            },
```

```
            {
                "match": {
                    "uri": "/admin/*"
                },
                "action": {
                    "pass": "applications/admin"
                }
            }
        ]
    }
}
```

Discussion

This recipe demonstrates a couple of basic route objects. The route object has two attributes: `match` and `action`. The `match` condition object takes three options: `host`, `method`, and `uri`. When specifying multiple options together, the match is evaluated as a logical AND. All of the `match` options accept either a string or an array of strings. When an array of strings is used for one of these options, the match at the option level is evaluated as a logical OR. Wildcards (`*`) and negations (`!`) are also supported. The patterns must be an exact match to the request. The route objects are evaluated in order, and the first match takes action. If no route is matched, an HTTP 404 is served.

The `action` attribute accepts an object value. Currently the only attribute of the `action` object is `pass`. The `pass` attribute defines the application to which the request should be directed. If only the `action` attribute, but no `match` condition, is specified in a route, requests are unconditionally directed to the `pass` value.

Additional Resources

Route Object

CHAPTER 4

Usage and Operations

Introduction

Understanding how to start and stop the NGINX Unit server, and the applications it runs, is essential. In this chapter you will learn how to start and stop the Unit service on **init.d** and **systemd** service managers, as well as how to start the Unit server in the foreground. This chapter also details how to submit the configuration objects to the Unit control API in order to start serving the application.

Startup and Shutdown

Problem

You need to start or stop the NGINX Unit server.

Solution

When Unit is installed through a repository, a startup file for a service manager such as, **init.d** or **systemd** is also installed and configured. These service managers will start Unit as a daemon.

Start Unit on an **init.d** system:

```
sudo /etc/init.d/unit start
```

Stop Unit on an **init.d** system:

```
sudo /etc/init.d/unit stop
```

Start Unit on a **systemd** system:

```
sudo systemclt start unit
```

Stop Unit on a **systemd** system:

```
sudo systemclt stop unit
```

Start Unit in the foreground. The following assumes that the Unit binary is installed into a directory defined in your `PATH`:

```
sudo unitd --no-daemon
```

Discussion

The service manager used to start the Unit daemon depends on the type of system it's running on. Each service manager has its own syntax for starting and stopping services. The service managers will start Unit as a daemon. An example of starting Unit in the foreground is also shown. This can be useful for testing, or when running Unit in a Docker container.

Applying Configuration

Problem

You need to start an application within NGINX Unit.

Solution

For this section, it's important to understand that the Unit configuration is represented as a single JSON object. Portions of the object can be interacted with in a RESTful manner. The following will be examples of working with specific application and listener objects, and then with the Unit config as a whole.

Locate the Unit control socket; example output is provided. The default value found in this example, `/var/run/control.unit.sock`, will be used throughout the book. As the control socket is owned by root by default, all `curl` commands will be run with `sudo`.

```
unitd -h

unit options:

  --version             print unit version and configure options
```

```
  --no-daemon           run unit in non-daemon mode

  --control ADDRESS     set address of control API socket
                        default: "unix:/var/run/control.unit.sock"
  ...
  ...
```

Create an application by submitting an application object to the control socket:

```
sudo curl -X PUT -d @/path/to/application-object.json \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config/applications/app-name
```

Configure a listener and direct it to the application:

This command removes all other listeners that might have been defined prior.

```
sudo curl -X PUT \
       -d '{"*:8080":{"pass":"applications/app-name"}}' \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config/listeners
```

You alternatively can create both objects at once by defining both the application and the listener object in one configuration file. Create a file named *php-app.json*:

```
{
    "listeners": {
        "*:8080": {
            "pass": "applications/app-name"
        }
    },
    "applications": {
        "app-name": {
            "type": "php",
            "processes": 20,
            "root": "/var/www/app/",
            "index": "index.php"
        }
    }
}
```

Submit the *php-app.json* configuration to the NGINX Unit control socket:

This command removes all other listeners, apps, and routes, that might have been defined prior.

```
sudo curl -X PUT -d @php-app.json  \
       --unix-socket /var/run/control.unit.sock \
       http://localhost/config/
```

Test your application:

```
curl localhost:8080
```

Discussion

All interactions with Unit are done through the control interface. The API is RESTful; applications and configurations are created, altered, or deleted through the API. In the examples for this solution, we build on the examples from Chapter 3 by submitting them to the Unit control interface. The alternate approach is to define all applications, routes, and listener objects in one JSON file and submit them together to the Unit configuration.

CHAPTER 5

Security

Introduction

Security is everyone's job. With NGINX Unit, security is at the forefront of the server's design. Unit naturally separates applications by spawning separate processes for each one, enabling isolation at the process and memory layer. Each application process can be owned by separate users, enabling security at the file permission layer as well. Finally, NGINX Unit has full SSL/TLS support, which enables Unit to serve applications through encrypted HTTPS communication. All of these security attributes are demonstrated in this chapter.

Application Isolation

Problem

You need to serve multiple applications and would like them to be fully isolated.

Solution

Configure the applications separately in NGINX Unit. Unit creates separate processes for each application, enabling isolation.

```
{
    "applications": {
        "auth-service": {
            "type": "php",
            "processes": 10,
```

```
            "root": "/var/www/auth/",
            "index": "index.php"
        },
        "key-service": {
            "type": "external",
            "processes": 2,
            "executable": "/var/key-service"
        }
    }
}
```

Discussion

NGINX Unit comprises two main processes and the application processes. The controller process serves the API interface used to configure Unit. The router process handles incoming requests and queues them for the application defined by the listener configuration. Each application is run as a separate process or group of processes.

Unix User Permissions

Problem

You need to further isolate your applications by using user permissions.

Solution

Use a different system user for each application so that Unit spawns the processes as separate users with their own permissions.

```
{
    "applications": {
        "auth-service": {
            "type": "php",
            ...
            "user": "auth-app"
        },
        "key-service": {
            "type": "external",
            ...
            "user": "key-app"
        }
    }
}
```

Discussion

Unit runs each application as a separate process or group of processes, enabling it to run these processes as separate system users. When configuring an application in Unit, there is an attribute for user and group. Using separate system users for each application will provide your applications with further isolation.

API Security through Encryption

Problem

You need to secure your application's communication with SSL/TLS certificates.

Solution

Create a `.pem` file that includes your certificate chain and private key:

```
cat cert.pem ca.pem key.pem | sudo tee bundle.pem > /dev/null
```

Upload the *bundle.pem* file created in the last step to Unit's certificate storage under a suitable name:

```
sudo curl -X PUT --data-binary @bundle.pem \
          --unix-socket /var/run/control.unit.socket \
          http://localhost/certificates/certificate-name
```

Configure a listener object to use the certificate. In this example, a file with the object will be written to a file named *tls-listener.json* for clarity:

```
{
    "*:8443": {
        "pass": "applications/app-name",
        "tls": {
            "certificate": "certificate-name"
        }
    }
}
```

Submit the *tls-listener.json* configuration to the Unit API:

This command removes all other listeners that might have been defined prior.

```
sudo curl -X PUT -d @tls-listener.json \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config/listeners
```

Validate that your application is communicating over TLS:

```
curl -v https://localhost:8443
```

Discussion

This recipe concatenates the certificate, certificate authority chain, and key into a bundle that can be used by NGINX Unit. After the certificate is uploaded to Unit's certificate store, it can be referenced by listeners. A listener object is constructed that references the IP and port on which to accept requests and that references the application via the `pass` attribute, as well as the certificate bundle object. The listener object is then submitted to the Unit control interface.

Validating that the TLS certificate is configured properly can be done by making a request to the listener. Using the verbose flag, `-v`, when issuing the `curl` command will print the TLS handshake operations if the certificate is configured properly.

Additional Resources

TLS Object

CHAPTER 6

Application Integration

Introduction

To provide examples of serving real-world applications with NGINX Unit, this chapter will demonstrate step-by-step setups of some common application frameworks. In this chapter you will learn how to serve WordPress, a common PHP content management system. You will also learn how to serve applications based in common frameworks such as Django (a Python framework) and Express (a Node.js framework). This chapter will demonstrate how to install applications onto a system and ensure that they have the correct file permissions and the configuration of NGINX Unit needed to serve them.

WordPress

Problem

You need to run WordPress with NGINX Unit.

Solution

To install WordPress, if you haven't already done so, check prerequisites to ensure that you have the necessary requirements. Next, configure the WordPress database. Then download and extract the WordPress files:

```
sudo mkdir /var/app/
sudo cd /var/app/
```

```
sudo wget https://wordpress.org/latest.tar.gz
sudo tar xzvf latest.tar.gz
```

In this example, the WordPress files will be stored in */var/app/wordpress/*.

Update the *wp-config.php* file with your database settings and other customizations.

Set the user file permissions for WordPress to ensure that the user that owns the PHP processes and the NGINX web server is able to access the files:

```
sudo chown -R wpuser:www-data /var/app/wordpress/
sudo find /var/app/wordpress/ -type d -exec chmod g+s {} \;
sudo chmod g+w /var/app/wordpress/wp-content
sudo chmod -R g+w /var/app/wordpress/wp-content/themes
sudo chmod -R g+w /var/app/wordpress/wp-content/plugins
```

Configure a PHP application object, as well as a listener object, and submit both objects to the NGINX Unit control interface. This example will configure two applications and listeners in order to isolate the main WordPress entry point, *index.php*, from the rest of the PHP files that can be run, such as *wp-admin.php*. Name the following JSON file *wordpress-unit.json*:

```
{
    "listeners": {
        "127.0.0.1:8090": {
            "pass": "applications/wp_index"
        },

        "127.0.0.1:8091": {
            "pass": "applications/wp_direct"
        }
    },

    "applications": {
        "wp_index": {
            "type": "php",
            "user": "wpuser",
            "group": "www-data",
            "root": "/var/app/wordpress/",
            "script": "index.php"
        },

        "wp_direct": {
            "type": "php",
            "user": "wpuser",
            "group": "www-data",
```

```
            "root": "/var/app/wordpress/",
            "index": "index.php"
        }
    }
}
```

Submit the *wordpress-unit.json* file to the Unit control interface:

```
sudo curl -X PUT -d @wordpress-unit.json \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config
```

Configure NGINX to serve static content and proxy requests to the applications that were just configured in NGINX Unit with the following basic upstream and server configuration:

```
upstream unit_wp_index {
    server 127.0.0.1:8090;
}

upstream unit_wp_direct {
    server 127.0.0.1:8091;
}

server {
    listen      80;
    server_name localhost;
    root        /var/app/wordpress/;

    location / {
        try_files $uri @index_php;
    }

    location @index_php {
        proxy_pass       http://unit_wp_index;
        proxy_set_header Host $host;
    }

    location /wp-admin {
        index index.php;
    }

    location ~* \.php$ {
        try_files        $uri =404;
        proxy_pass       http://unit_wp_direct;
        proxy_set_header Host $host;
    }
}
```

Use a browser to make a request to the NGINX server on port `80`, and finish the installation process.

Discussion

In this recipe, WordPress is installed from scratch. The system and database first need to be prepared to the WordPress specifications. After the system is prepared, the code base is downloaded and unpacked to a location on the filesystem.

Once the application code is on the filesystem, WordPress needs to be informed how to connect to the database. This is done by altering a configuration file that is included in the code base. For the sake of brevity, this is statically configured. In a production system, environment variables would be used and set when configuring the Unit application.

After the database connection has been configured, the file permissions are changed so that the system user that will own the Unit processes will be able to read the files. Permissions are also set for the system group, which will be used by the NGINX process to serve the static content.

When configuring Unit to serve the application, security precautions are taken to isolate the main WordPress entry point from direct PHP files. The main difference between the two configured applications is the `script` and `index` settings. When the `script` application attribute is used, Unit will run any URL it receives, such as */wp-login.php*. When the `index` attribute is used, only the file provided as the value will be executed. This separation allows for application settings that vary based on the intended usage. A useful example would be restricting access, or allowing for a longer request time-out for the administration section of WordPress.

Unit does not serve static content, of which WordPress has a lot. To serve this content, the NGINX web server is needed. In order to provide a single endpoint for both the static and dynamic content, NGINX is also configured to proxy to the applications. The provided configuration hosts a web server on port `80`. When a request is made to the web server, NGINX will check the filesystem for a static file matching the URL. If the file is not found, the request is proxied to the `wp-index` Unit listener (unless the URL ends with `.php`, in which case the request will be proxied to the `wp-direct` Unit listener).

Additional Resources

WordPress How To

Django

Problem

You have a Python Django application you want to serve with NGINX Unit.

Solution

Prepare your existing project or create a new one. In this example, the source code will be placed in */var/project/*. Start by ensuring that the correct file permissions are set:

```
sudo chown -R app-user:app-user /var/project/
```

Detailing the directory structure of the example is important, because Unit needs to know how to import the WSGI module in order to run the application. Thus the Unit application object values depend on the directory structure:

```
/var/project/
├── manage.py
├── app1/
│   └── ...
├── app2/
│   └── ...
└── project/
    ├── ...
    └── wsgi.py
```

Construct an NGINX Unit Python application object and associated listener. Name this file *django-unit.json*:

```
{
    "listeners": {
        "127.0.0.1:8080": {
            "pass": "applications/django_project"
        }
    },

    "applications": {
        "django_project": {
            "type": "python",
            "path": "/var/project/",
            "home": "/path/to/virtual-env/if/used/",
```

```
            "module": "project.wsgi",
            "user": "app-user"
        }
    }
}
```

Submit the *django-unit.json* file to the Unit control interface:

```
sudo curl -X PUT -d @django-unit.json \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config
```

Validate that the application is running by making a request to the server on port `8080`:

```
curl http://localhost:8080
```

Discussion

In this recipe, a Django project is served with NGINX Unit. For Unit to be permitted to read the files, the correct file permissions need to be set. In the example, the files are owned by the system user that will be running the application.

This recipe shows the directory structure, not because it needs to be followed but because it shows how the `module` attribute of the Unit application object for Python applications is configured. The value of the `module` attribute is used to import the WSGI module, with standard Python import syntax, from the directory specified by the `path` attribute.

The Unit configuration specifies that this application is of type `python`. As the version of Python is not specified, the latest version is used. The `path` attribute specifies the path to the base directory of the application. If a virtual environment is being used, the optional `home` attribute can be set to the base directory of the virtual environment. Unit imports the WSGI object by use of the `module` attribute and runs the application as specified by the system user. The configuration then defines a listener object that instructs Unit to send incoming requests on the `127.0.0.1:8080` interface, to be directed to the `django_project` application.

Additional Resources

Django How To

Express

Problem

You have a Node.js application that utilizes the Express framework.

Solution

Set up your project and ensure that Node is installed.

To run Node applications in NGINX Unit, an NPM package is required. The version of the NPM package `unit-http` must match the version of NGINX Unit being used. It's wise to version-lock the Unit server and NPM package to avoid version conflicts. To build and install the NPM package, you will also need the development Unit package, which includes necessary header files. The development package was included in the installation process in Chapter 2:

```
npm install unit-http
```

Unit will call the Node application's entry point as an executable. Add the following line to the beginning of the entry point file:

```
#!/usr/bin/env node
```

Make the entry point executable, and ensure that it's owned by the system user that will run the application. In the example, the entry point file is *index.js*, and the project directory is */var/app/*:

```
chown -R app-user /var/app/
chmod u+x index.js
```

To serve an Express application with Unit, the code needs to be slightly modified. The default Express HTTP server, `ServerResponse`, and `IncomingMessage` objects need to be replaced with objects from the default `http` package to the `unit-http` package. The following "Hello World!" example shows how to rewire the application:

```
#!/usr/bin/env node

const {
  createServer,
  IncomingMessage,
  ServerResponse,
} = require('unit-http')

require('http').ServerResponse = ServerResponse
```

```
require('http').IncomingMessage = IncomingMessage

const express = require('express')

const app = express()

app.get('/', (req, res) => {
  res.set('X-Header-Example', 'Value')
  res.send('Hello, Unit!')
})

createServer(app).listen()
```

Construct the NGINX Unit application and listener objects for this project and name the file *express-unit.json*:

```
{
    "listeners": {
        "127.0.0.1:8080": {
            "pass": "applications/express_project"
        }
    },

    "applications": {
        "express_project": {
            "type": "external",
            "executable": "/var/app/index.js",
            "user": "app-user"
        }
    }
}
```

Submit the *express-unit.json* file to the Unit control interface:

```
sudo curl -X PUT -d @express-unit.json \
       --unix-socket /var/run/control.unit.socket \
       http://localhost/config
```

Validate that the application is running by making a request to the server on port `8080`.

Discussion

In this recipe, the `unit-http` package is installed to the project, and its objects are used rather than the default `http` server objects. The entry point file is made executable and the correct file permissions are set on the project, so that Unit is able to read the modules and run the entry point. Lastly, the Unit application and listener objects are constructed and submitted to the Unit control API. The `executable` attribute specifies the location of the entry point file. An

optional application object attribute for external application types, named `arguments`, can be used if there are arguments that need to be passed to the executable.

Additional Resources

Express How To

CHAPTER 7

Ecosystem Integration

Introduction

Throughout this chapter you will learn about operational integration as it pertains to NGINX Unit. Unit applications may need to be served via an NGINX proxy or load balancer, to which the configuration will be detailed. Also included are recipes that enable you to securely expose the Unit control interface through NGINX. Other topics include running Unit within a container and deploying application version upgrades through the control API.

Reverse Proxying to Unit Applications through NGINX

Problem

You need to serve an application running in NGINX Unit through a NGINX server acting as a reverse proxy or load balancer.

Solution

Configure an `upstream` block in the NGINX configuration made up of Unit servers:

```
upstream unit_backend {
    server 127.0.0.1:8080; # Local Reverse Proxy
    server 10.0.0.12:8080; # Remote Server Load Balance
    server 10.0.1.12:8080; # Remote Server Load Balance
}
```

Configure a `server` block within the NGINX configuration to proxy requests to the `upstream` server set:

```
server {
    # Typical NGINX server setup and security directives

    location / {
        # NGINX Proxy Settings
        proxy_pass http://unit_backend;
    }
}
```

Discussion

The NGINX web server and reverse proxy load balancer is a fully dynamic application gateway. It can be used as a web server, reverse proxy, load balancer, and more. For brevity, this recipe assumes that the NGINX `server` block has been configured with the necessary required and security-concerned directives.

In a reverse proxy situation, the NGINX server would be configured on the same physical or virtual machine as NGINX Unit. The `upstream` block would be configured with a `server` directive with a parameter specifying the same interface configured for the Unit listener object. In this example, the localhost `127.0.0.1` is used, in conjunction with the port `8080`.

In a load balancing situation, the NGINX server would be configured with an `upstream` block that contains multiple remote `server` directives. The example provides two `server` directives specifying different remote NGINX Unit servers at IP addresses `10.0.0.12` and `10.0.1.12`. Both of these Unit servers would be configured with listener objects on port `8080` for the same application.

This example further demonstrates how a properly configured `server` block can receive connections and direct the request to the application defined by the `upstream` block. This is done by defining a `location` block and using the `proxy_pass` directive with a parameter that specifies the protocol and destination. In this example, the destination is the `upstream` server block, named `unit_backend`.

Incoming connections to the NGINX server will be processed, and requests matching the configured server definition will be directed to the configuration within this `server` block. In this example, all configuration requests will be sent to the NGINX Unit server for

processing. The NGINX Unit server will return the request to the NGINX server, which will return the request to the client.

Additional Resources

NGINX Integration

Securely Serving the NGINX Unit Control API

Problem

You would like to remotely and securely configure the Unit application server.

Solution

Configure a NGINX reverse proxy to the control interface Unix socket. Ensure that it is only available internally and that client-server encryption is enforced.

```
server {

    # Configure SSL encryption
    server 443 ssl;
    ssl_certificate /path/to/ssl/cert.pem;
    ssl_certificate_key /path/to/ssl/cert.key;

    # Configure SSL client certificate validation
    ssl_client_certificate /path/to/ca.pem;
    ssl_verify_client on;

    # Configure network ACLs
    #allow 1.2.3.4; # Uncomment and update with the IP addresses
                    # and networks of your administrative systems.
    deny all;

    # Configure HTTP Basic authentication
    auth_basic on;
    auth_basic_user_file /path/to/htpasswd;

    location / {
        proxy_pass http://unix:/var/run/control.unit.sock;
    }
}
```

Discussion

This recipe configures the NGINX reverse proxy server to serve the NGINX Unit control interface through an HTTPS connection. The NGINX server is configured to serve only on port `443` and to only accept encrypted connections. The SSL/TLS directives of the NGINX server must be configured to specify a given certificate and key for encryption. This configuration also requires the client to provide a certificate signed by the specified certificate authority as a means of authentication. For further security, the configuration denies all requests from any client IP that is not specified by the `allow` directive. The `allow` directive must be uncommented and configured to your internal IP or CIDR. Finally, a username and password must be specified via HTTP basic auth. The `auth_basic_user_file` directive defines a file that contains usernames and hashed passwords of authorized users.

Once all security measures are met, NGINX will proxy the request to the NGINX Unit control interface. By default, the Unit control interface listens on a Unix socket. The system user running NGINX must have permission to read and write to this Unix socket file.

Additional Resources

NGINX Integration

Containerized Environment

Problem

You would like to use NGINX Unit as a middleware server in a containerized environment.

Solution

Build a unit configuration file at the base of the project. Name the file *unit-conf.json*:

```
{
    "listeners": {
        "*:8080": {
            "pass": "applications/php_project"
        }
    },
    "applications": {
```

```
        "php_project": {
            "type": "php",
            "processes": 1,
            "root": "/var/app",
            "index": "index.php"
        }
    }
}
```

Use the Official NGINX Unit Docker Image as the base. Create a Dockerfile with the following:

```
FROM nginx/unit

ADD / /var/app/

ADD /unit-conf.json /var/lib/unit/conf.json
```

Build the Dockerfile into an image:

```
docker build -t unit-example .
```

Run the Docker image and expose the listener through the Docker proxy for testing. The following example uses the Docker `-p` flag to configure a proxy, exposing port `8080` proxied to port `8080`. As a reminder, the port number before the `:` is the port exposed on the local machine:

```
docker run -p 8080:8080 unit-example
```

Make a request to the exposed Docker proxy to validate:

```
curl localhost:8080
```

Discussion

This recipe demonstrates the basics of using NGINX Unit as a middleware server for dockerized applications. A Unit configuration file is created for the application. A Dockerfile is then crafted, based on the Official NGINX Unit Docker Image. Within the Dockerfile, the application code is added to the image. The configuration file is then added to the image in the location of the Unit state file. This ensures that Unit will start with the application and listener objects configured.

The Dockerfile is then built, rendering an image tagged `unit-example`. The Docker image is then run with the proxy flag to expose the listener to the host. Once running, the Docker container is validated.

Furthermore, with Docker you are able to mount volumes with the `-v` flag. By doing so you are able to expose the host's filesystem. If the control interface is overridden via the `CMD` directive in the Dockerfile, and exposed by the Docker proxy, remote reconfiguration of the Unit container is enabled. In this configuration it is possible to add applications that exist on the host's filesystem and to reconfigure Unit listeners to serve these applications remotely through the control API. This technique may be helpful for local development environments.

Additional Resources

Unit in Docker

Deployments

Problem

You need to deploy a new version of an application without downtime.

Solution

Utilize NGINX Unit's API to switch between application versions through an API call. This recipe will use a directory structure laid out in the following way:

```
/var/app/
├── version-1
│   ├── index.php
│   └── ...
└── version-2
    ├── index.php
    └── ...
```

The current state of the Unit configuration is as such:

```
{
    "listeners": {
        "*:8080": {
            "pass": "applications/php_project_version_1"
        }
    },
    "applications": {
        "php_project_version_1": {
           "type": "php",
           "processes": 2,
```

```
            "root": "/var/app/version-1",
            "index": "index.php"
        }
    }
}
```

Create another file named *php-v2.json* file with the following JSON:

```
{
     "type": "php",
     "processes": 2,
     "root": "/var/app/version-2",
     "index": "index.php"
}
```

Make an API call to the control interface. Provide the *php-v2.json* as the JSON body. Use the RESTful syntax to name the Unit application `php_project_version_2`:

```
sudo curl -X PUT -d @php-v2.json \
      --unix-socket /var/run/control.unit.sock \
      http://localhost/config/applications/php_project_version_2
```

Make the following request to the Unit control interface to validate that both applications are configured:

```
sudo curl --unix-socket /var/run/control.unit.sock \
      http://localhost/config
{
        "listeners": {
                "*:8080": {
                        "pass": "applications/php_project_version_1"
                }
        },

        "applications": {
                "php_project_version_1": {
                        "type": "php",
                        "processes": 2,
                        "root": "/var/app/version-1",
                        "index": "index.php"
                },

                "php_project_version_2": {
                        "type": "php",
                        "processes": 2,
                        "root": "/var/app/version-2",
                        "index": "index.php"
                }
        }
}
```

Make a request to the control interface with the following command, instructing Unit to switch the listener `*:8080` to point to the `php_project_version_2` application:

```
sudo curl -X PUT -d '"php_project_version_2"' \
      --unix-socket /var/run/control.unit.sock \
      'http://localhost/config/listeners/*:8080/application'
```

Make the following request to the Unit control interface to validate that the listener has been reconfigured to direct requests to the `php_project_version_2` application:

```
sudo curl --unix-socket /var/run/control.unit.sock \
      http://localhost/config
{
    "listeners": {
        "*:8080": {
            "pass": "applications/php_project_version_2"
        }
    },

    "applications": {
        "php_project_version_1": {
            "type": "php",
            "processes": 2,
            "root": "/var/app/version-1",
            "index": "index.php"
        },

        "php_project_version_2": {
            "type": "php",
            "processes": 2,
            "root": "/var/app/version-2",
            "index": "index.php"
        }
    }
}
```

Make a request to the control interface to remove the `php_project_version_1` application:

```
sudo curl -X DELETE \
      --unix-socket /var/run/control.unit.sock \
      http://localhost/config/applications/php_project_version_1
```

Discussion

This recipe demonstrates the deployment of a new version of an application. The example starts from a pre-configured state, with a single application version being served on port `8080`. NGINX Unit is

then configured to start another application of a new version. Both versions run in parallel as separate process sets. Unit is then instructed to route incoming requests to the new application version. Finally, the older application version is removed, and the processes that served that application are removed.

About the Author

Derek DeJonghe has had a lifelong passion for technology. His background and experience in web development, system administration, and networking give him a well-rounded understanding of modern web architecture. Derek currently manages a cloud consulting firm specializing in cloud native application development, as well as Infrastructure, configuration, and CI/CD pipelines as code. A focus of Derek's work has been on moving the pets versus cattle analogy from single servers to entire environments, enabling teams to build and destroy environments at will for integration testing. With a proven track record for resilient cloud architecture, Derek helps RightBrain Networks be one of the strongest cloud consulting agencies and managed service providers in partnership with AWS today.