Red Hat Enterprise Linux 9

Installing and using dynamic programming languages

Installing and using Python and PHP in Red Hat Enterprise Linux 9

Last Updated: 2025-03-28

Red Hat Enterprise Linux 9 Installing and using dynamic programming languages

Installing and using Python and PHP in Red Hat Enterprise Linux 9

Legal Notice

Abstract

Install and use Python 3, package Python 3 RPMs, and learn how to handle interpreter directives in Python scripts. Install the PHP scripting language, use PHP with the Apache HTTP Server or the ngninx web server, and run a PHP script from a command-line interface.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION 3

CHAPTER 1. INTRODUCTION TO PYTHON 4
1.1. PYTHON VERSIONS 4
1.2. MAJOR DIFFERENCES IN THE PYTHON ECOSYSTEM SINCE RHEL 8 4

CHAPTER 2. INSTALLING AND USING PYTHON 6
2.1. INSTALLING PYTHON 3 6
2.2. INSTALLING ADDITIONAL PYTHON 3 PACKAGES 6
2.3. INSTALLING ADDITIONAL PYTHON 3 TOOLS FOR DEVELOPERS 7
2.4. USING PYTHON 8

CHAPTER 3. PACKAGING PYTHON 3 RPMS 10
3.1. SPEC FILE DESCRIPTION FOR A PYTHON PACKAGE 10
3.2. COMMON MACROS FOR PYTHON 3 RPMS 12
3.3. USING AUTOMATICALLY GENERATED DEPENDENCIES FOR PYTHON RPMS 13

CHAPTER 4. HANDLING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS 15
4.1. MODIFYING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS 15

CHAPTER 5. INSTALLING TCL/TK 17
5.1. INTRODUCTION TO TCL/TK 17
5.2. INSTALLING TCL 17
5.3. INSTALLING TK 17

CHAPTER 6. USING THE PHP SCRIPTING LANGUAGE 19
6.1. INSTALLING THE PHP SCRIPTING LANGUAGE 19
6.2. USING THE PHP SCRIPTING LANGUAGE WITH A WEB SERVER 19
6.2.1. Using PHP with the Apache HTTP Server 20
6.2.2. Using PHP with the nginx web server 21
6.3. RUNNING A PHP SCRIPT USING THE COMMAND LINE 23
6.4. ADDITIONAL RESOURCES 24

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

1. Log in to the Jira website.
2. Click **Create** in the top navigation bar
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. INTRODUCTION TO PYTHON

Python is a high-level programming language that supports multiple programming paradigms, such as object-oriented, imperative, functional, and procedural paradigms. Python has dynamic semantics and can be used for general-purpose programming.

With Red Hat Enterprise Linux, many packages that are installed on the system, such as packages providing system tools, tools for data analysis, or web applications, are written in Python. To use these packages, you must have the **python*** packages installed.

1.1. PYTHON VERSIONS

Python 3.9 is the default **Python** implementation in RHEL 9. **Python 3.9** is distributed in a non-modular **python3** RPM package in the BaseOS repository and is usually installed by default. **Python 3.9** will be supported for the whole life cycle of RHEL 9.

Additional versions of **Python 3** are distributed as non-modular RPM packages with a shorter life cycle through the AppStream repository in minor RHEL 9 releases. You can install these additional **Python 3** versions in parallel with Python 3.9.

Python 2 is not distributed with RHEL 9.

Table 1.1. Python versions in RHEL 9

Version	Package to install	Command examples	Available since	Life cycle
Python 3.9	**python3**	**python3**, **pip3**	RHEL 9.0	full RHEL 9
Python 3.11	**python3.11**	**python3.11**, **pip3.11**	RHEL 9.2	shorter
Python 3.12	**python3.12**	**python3.12**, **pip3.12**	RHEL 9.4	shorter

For details about the length of support, see Red Hat Enterprise Linux Life Cycle and Red Hat Enterprise Linux Application Streams Life Cycle.

1.2. MAJOR DIFFERENCES IN THE PYTHON ECOSYSTEM SINCE RHEL 8

The following are the major changes in the Python ecosystem in RHEL 9 compared to RHEL 8:

The unversioned python command

The unversioned form of the **python** command (**/usr/bin/python**) is available in the **python-unversioned-command** package. On some systems, this package is not installed by default. To install the unversioned form of the **python** command manually, use the **dnf install /usr/bin/python** command.

In RHEL 9, the unversioned form of the **python** command points to the default **Python 3.9** version and it is an equivalent to the **python3** and **python3.9** commands. In RHEL 9, you cannot configure the unversioned command to point to a different version than **Python 3.9**.

The **python** command is intended for interactive sessions. In production, it is recommended to use **python3**, **python3.9**, **python3.11**, or **python3.12** explicitly.

You can uninstall the unversioned **python** command by using the **dnf remove /usr/bin/python** command.

If you need a different **python** or **python3** command, you can create custom symlinks in **/usr/local/bin** or **~/.local/bin**, or use a Python virtual environment.

Several other unversioned commands are available, such as **/usr/bin/pip** in the **python3-pip** package. In RHEL 9, all unversioned commands point to the default **Python 3.9** version.

Architecture-specific Python wheels

Architecture-specific Python **wheels** built on RHEL 9 newly adhere to the upstream architecture naming, which allows customers to build their Python **wheels** on RHEL 9 and install them on non-RHEL systems. Python **wheels** built on previous releases of RHEL are compatible with later versions and can be installed on RHEL 9. Note that this affects only **wheels** containing Python extensions, which are built for each architecture, not Python **wheels** with pure Python code, which is not architecture-specific.

CHAPTER 2. INSTALLING AND USING PYTHON

In RHEL 9, **Python 3.9** is the default **Python** implementation. Since RHEL 9.2, **Python 3.11** is available as the **python3.11** package suite, and since RHEL 9.4, **Python 3.12** as the **python3.12** package suite.

The unversioned **python** command points to the default **Python 3.9** version.

2.1. INSTALLING PYTHON 3

The default **Python** implementation is usually installed by default. To install it manually, use the following procedure.

Procedure

- To install **Python 3.9**, use:

```
# dnf install python3
```

- To install **Python 3.11**, use:

```
# dnf install python3.11
```

- To install **Python 3.12**, use:

```
# dnf install python3.12
```

Verification

- To verify the **Python** version installed on your system, use the **--version** option with the **python** command specific for your required version of **Python**.
- For **Python 3.9**:

```
$ python3 --version
```

- For **Python 3.11**:

```
$ python3.11 --version
```

- For **Python 3.12**:

```
$ python3.12 --version
```

2.2. INSTALLING ADDITIONAL PYTHON 3 PACKAGES

Packages prefixed with **python3-** contain add-on modules for the default **Python 3.9** version. Packages prefixed with **python3.11-** contain add-on modules for **Python 3.11**. Packages prefixed with **python3.12-** contain add-on modules for **Python 3.12**.

Procedure

- To install the **Requests** module for **Python 3.9**, use:

```
# dnf install python3-requests
```

- To install the **pip** package installer from **Python 3.9**, use:

```
# dnf install python3-pip
```

- To install the **pip** package installer from **Python 3.11**, use:

```
# dnf install python3.11-pip
```

- To install the **pip** package installer from **Python 3.12**, use:

```
# dnf install python3.12-pip
```

Additional resources

- Upstream documentation about Python add-on modules

2.3. INSTALLING ADDITIONAL PYTHON 3 TOOLS FOR DEVELOPERS

Additional **Python** tools for developers are distributed mostly through the CodeReady Linux Builder (CRB) repository.

The **python3-pytest** package and its dependencies are available in the AppStream repository.

The CRB repository contains, for example, the following packages:

- **python3*-idle**
- **python3*-debug**
- **python3*-Cython**
- **python3.11-pytest** and its dependencies
- **python3.12-pytest** and its dependencies.

IMPORTANT

The content in the CodeReady Linux Builder repository is unsupported by Red Hat.

NOTE

Not all upstream **Python**-related packages are available in RHEL.

To install packages from the CRB repository, use the following procedure.

Procedure

1. Enable the CodeReady Linux Builder repository:

```
# subscription-manager repos --enable codeready-builder-for-rhel-9-x86_64-rpms
```

2. Install the **python3*-Cython** package:
   - For **Python 3.9**:

     ```
     # dnf install python3-Cython
     ```

   - For **Python 3.11**:

     ```
     # dnf install python3.11-Cython
     ```

   - For **Python 3.12**:

     ```
     # dnf install python3.12-Cython
     ```

Additional resources

- How to enable and make use of content within CodeReady Linux Builder
- Package manifest

2.4. USING PYTHON

The following procedure contains examples of running the **Python** interpreter or **Python**-related commands.

Prerequisites

- Ensure that **Python** is installed.
- If you want to download and install third-party applications for **Python 3.11** or **Python 3.12**, install the **python3.11-pip** or **python3.12-pip** package.

Procedure

- To run the **Python 3.9** interpreter or related commands, use, for example:

  ```
  $ python3
  $ python3 -m venv --help
  $ python3 -m pip install package
  $ pip3 install package
  ```

- To run the **Python 3.11** interpreter or related commands, use, for example:

  ```
  $ python3.11
  $ python3.11 -m venv --help
  $ python3.11 -m pip install package
  $ pip3.11 install package
  ```

- To run the **Python 3.12** interpreter or related commands, use, for example:

```
$ python3.12
$ python3.12 -m venv --help
$ python3.12 -m pip install package
$ pip3.12 install package
```

CHAPTER 3. PACKAGING PYTHON 3 RPMS

You can install Python packages on your system either from the upstream PyPI repository using the **pip** installer, or using the DNF package manager. DNF uses the RPM package format, which offers more downstream control over the software.

The packaging format of native Python packages is defined by Python Packaging Authority (PyPA) Specifications. Most Python projects use the **distutils** or **setuptools** utilities for packaging, and defined package information in the **setup.py** file. However, possibilities of creating native Python packages have evolved over time. For more information about emerging packaging standards, see pyproject-rpm-macros.

This chapter describes how to package a Python project that uses **setup.py** into an RPM package. This approach provides the following advantages compared to native Python packages:

- Dependencies on Python and non-Python packages are possible and strictly enforced by the **DNF** package manager.
- You can cryptographically sign the packages. With cryptographic signing, you can verify, integrate, and test content of RPM packages with the rest of the operating system.
- You can execute tests during the build process.

3.1. SPEC FILE DESCRIPTION FOR A PYTHON PACKAGE

A SPEC file contains instructions that the **rpmbuild** utility uses to build an RPM. The instructions are included in a series of sections. A SPEC file has two main parts in which the sections are defined:

- Preamble (contains a series of metadata items that are used in the Body)
- Body (contains the main part of the instructions)

An RPM SPEC file for Python projects has some specifics compared to non-Python RPM SPEC files.

IMPORTANT

A name of any RPM package of a Python library must always include the **python3-**, **python3.11-**, or **python3.12-** prefix.

Other specifics are shown in the following SPEC file example for the **python3*-pello** package. For description of such specifics, see the notes below the example.

An example spec file for the pello program written in Python

```
%global python3_pkgversion 3.11                    1

Name:          python-pello                        2
Version:       1.0.2
Release:       1%{?dist}
Summary:       Example Python library

License:       MIT
URL:           https://github.com/fedora-python/Pello
Source:        %{url}/archive/v%{version}/Pello-%{version}.tar.gz
```

```
BuildArch:      noarch
BuildRequires:  python%{python3_pkgversion}-devel                    3

# Build dependencies needed to be specified manually
BuildRequires:  python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python%{python3_pkgversion}-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command line.}

%description %_description

%package -n python%{python3_pkgversion}-pello                        4
Summary:        %{summary}

%description -n python%{python3_pkgversion}-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build                                                           5


%install
# The macro only supported projects with setup.py
%py3_install


%check                                                               6
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

1. By defining the **python3_pkgversion** macro, you set which Python version this package will be built for. To build for the default Python version 3.9, either set the macro to its default value **3** or remove the line entirely.

2. When packaging a Python project into RPM, always add the **python-** prefix to the original name of the project. The original name here is **pello** and, therefore, the name of the Source RPM (SRPM) is **python-pello**.

3. The **BuildRequires** directive specifies what packages are required to build and test this package. In **BuildRequires**, always include items providing tools necessary for building Python packages: **python3-devel** (or **python3.11-devel** or **python3.12-devel**) and the relevant projects needed by the specific software that you package, for example, **python3-setuptools** (or **python3.11-setuptools** or **python3.12-setuptools**) or the runtime and testing dependencies needed to run the tests in the **%check** section.

4. When choosing a name for the binary RPM (the package that users will be able to install), add a versioned Python prefix. Use the **python3-** prefix for the default Python 3.9, the **python3.11-** prefix for Python 3.11, or the **python3.12-** prefix for Python 3.12. You can use the **%{python3_pkgversion}** macro, which evaluates to **3** for the default Python version 3.9 unless you set it to an explicit version, for example, **3.11** (see footnote 1).

5. The **%py3_build** and **%py3_install** macros run the **setup.py build** and **setup.py install** commands, respectively, with additional arguments to specify installation locations, the interpreter to use, and other details.

6. The **%check** section should run the tests of the packaged project. The exact command depends on the project itself, but it is possible to use the **%pytest** macro to run the **pytest** command in an RPM-friendly way.

3.2. COMMON MACROS FOR PYTHON 3 RPMS

In a SPEC file, always use the macros that are described in the following *Macros for Python 3 RPMs* table rather than hardcoding their values. You can redefine which Python 3 version is used in these macros by defining the **python3_pkgversion** macro on top of your SPEC file (see Section 3.1, "SPEC file description for a Python package"). If you define the **python3_pkgversion** macro, the values of the macros described in the following table will reflect the specified Python 3 version.

Table 3.1. Macros for Python 3 RPMs

Macro	Normal Definition	Description
%{python3_pkgversion}	3	The Python version that is used by all other macros. Can be redefined to **3.11** to use Python 3.11, or to **3.12** to use Python 3.12
%{python3}	/usr/bin/python3	The Python 3 interpreter
%{python3_version}	3.9	The major.minor version of the Python 3 interpreter
%{python3_sitelib}	/usr/lib/python3.9/site-packages	The location where pure-Python modules are installed

Macro	Normal Definition	Description
%{python3_sitearch}	/usr/lib64/python3.9/site-packages	The location where modules containing architecture-specific extension modules are installed
%py3_build		Runs the **setup.py build** command with arguments suitable for an RPM package
%py3_install		Runs the **setup.py install** command with arguments suitable for an RPM package
%{py3_shebang_flags}	s	The default set of flags for the Python interpreter directives macro, **%py3_shebang_fix**
%py3_shebang_fix		Changes Python interpreter directives to **#! %{python3}**, preserves any existing flags (if found), and adds flags defined in the **%{py3_shebang_flags}** macro

Additional resources

- Python macros in upstream documentation

3.3. USING AUTOMATICALLY GENERATED DEPENDENCIES FOR PYTHON RPMS

The following procedure describes how to use automatically generated dependencies when packaging a Python project as an RPM.

Prerequisites

- A SPEC file for the RPM exists. For more information, see SPEC file description for a Python package.

Procedure

1. Make sure that one of the following directories containing upstream-provided metadata is included in the resulting RPM:
   - **.dist-info**
   - **.egg-info**
     The RPM build process automatically generates virtual **pythonX.Ydist** provides from these directories, for example:

     ```
     python3.9dist(pello)
     ```

The Python dependency generator then reads the upstream metadata and generates runtime requirements for each RPM package using the generated **pythonX.Ydist** virtual provides. For example, a generated requirements tag might look as follows:

```
Requires: python3.9dist(requests)
```

2. Inspect the generated requires.
3. To remove some of the generated requires, use one of the following approaches:
   a. Modify the upstream-provided metadata in the **%prep** section of the SPEC file.
   b. Use automatic filtering of dependencies described in the upstream documentation.
4. To disable the automatic dependency generator, include the **%{?python_disable_dependency_generator}** macro above the main package's **%description** declaration.

Additional resources

- Automatically generated dependencies

CHAPTER 4. HANDLING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS

In Red Hat Enterprise Linux 9, executable Python scripts are expected to use interpreter directives (also known as hashbangs or shebangs) that explicitly specify at a minimum the major Python version. For example:

```
#!/usr/bin/python3
#!/usr/bin/python3.9
#!/usr/bin/python3.11
#!/usr/bin/python3.12
```

The **/usr/lib/rpm/redhat/brp-mangle-shebangs** buildroot policy (BRP) script is run automatically when building any RPM package, and attempts to correct interpreter directives in all executable files.

The BRP script generates errors when encountering a Python script with an ambiguous interpreter directive, such as:

```
#!/usr/bin/python
```

or

```
#!/usr/bin/env python
```

4.1. MODIFYING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS

Use the following procedure to modify interpreter directives in Python scripts that cause build errors at RPM build time.

Prerequisites

- Some of the interpreter directives in your Python scripts cause a build error.

Procedure

- To modify interpreter directives, complete one of the following tasks:
  - Use the following macro in the **%prep** section of your SPEC file:

    ```
    # %py3_shebang_fix SCRIPTNAME …
    ```

    SCRIPTNAME can be any file, directory, or a list of files and directories.

    As a result, all listed files and all **.py** files in listed directories will have their interpreter directives modified to point to **%{python3}**. Existing flags from the original interpreter directive will be preserved and additional flags defined in the **%{py3_shebang_flags}** macro will be added. You can redefine the **%{py3_shebang_flags}** macro in your SPEC file to change the flags that will be added.

  - Apply the **pathfix.py** script from the **python3-devel** package:

    ```
    # pathfix.py -pn -i %{python3} PATH …
    ```

You can specify multiple paths. If a ***PATH*** is a directory, **pathfix.py** recursively scans for any Python scripts matching the pattern **^[a-zA-Z0-9_]+\.py$**, not only those with an ambiguous interpreter directive. Add the command above to the **%prep** section or at the end of the **%install** section.

- Modify the packaged Python scripts so that they conform to the expected format. For this purpose, you can use the **pathfix.py** script outside the RPM build process, too. When running **pathfix.py** outside an RPM build, replace **%{python3}** from the preceding example with a path for the interpreter directive, such as **/usr/bin/python3** or **/usr/bin/python3.11**.

Additional resources

- Interpreter invocation

CHAPTER 5. INSTALLING TCL/TK

5.1. INTRODUCTION TO TCL/TK

Tcl is a dynamic programming language, while **Tk** is a graphical user interface (GUI) toolkit. They provide a powerful and easy-to-use platform for developing cross-platform applications with graphical interfaces. As a dynamic programming language, 'Tcl' provides simple and flexible syntax for writing scripts. The **tcl** package provides the interpreter for this language and the C library. You can use **Tk** as GUI toolkit that provides a set of tools and widgets for creating graphical interfaces. You can use various user interface elements such as buttons, menus, dialog boxes, text boxes, and canvas for drawing graphics. **Tk** is the GUI for many dynamic programming languages.

For more information about Tcl/Tk, see the Tcl/Tk manual or Tcl/Tk documentation web page.

5.2. INSTALLING TCL

The default **Tcl** implementation is usually installed by default. To install it manually, use the following procedure.

Procedure

- To install **Tcl**, use:

```
# dnf install tcl
```

Verification

- To verify the Tcl version installed on your system, run the interpreter **tclsh**.

```
$ tclsh
```

- In the interpreter run this command:

```
% info patchlevel
8.6
```

- You can exit the interpreter interface by pressing **Ctrl**+**C**

5.3. INSTALLING TK

The default **Tk** implementation is usually installed by default. To install it manually, use the following procedure.

Procedure

- To install **Tk**, use:

```
# dnf install tk
```

Verification

- To verify the **Tk** version installed on your system, run the window shell **wish**. You need to be running a graphical display.

```
$ wish
```

- In the shell run this command:

```
% puts $tk_version
8.6
```

- You can exit the interpreter interface by pressing **Ctrl**+**C**

CHAPTER 6. USING THE PHP SCRIPTING LANGUAGE

Hypertext Preprocessor (PHP) is a general-purpose scripting language mainly used for server-side scripting. You can use PHP to run the PHP code by using a web server.

6.1. INSTALLING THE PHP SCRIPTING LANGUAGE

In RHEL 9, PHP is available in the following versions and formats:

- PHP 8.0 as the **php** RPM package
- PHP 8.1 as the **php:8.1** module stream
- PHP 8.2 as the **php:8.2** module stream

Procedure

Depending on your scenario, complete one of the following steps:

- To install PHP 8.0, enter:

```
# dnf install php
```

- To install the **php:8.1** or **php:8.2** module stream with the default profile, enter for example:

```
# dnf module install php:8.1
```

  The default **common** profile installs also the **php-fpm** package, and preconfigures PHP for use with the Apache HTTP Server or nginx.

- To install a specific profile of the **php:8.1** or **php:8.2** module stream, use for example:

```
# dnf module install php:8.1/profile
```

  Available profiles are as follows:

  - **common** - The default profile for server-side scripting using a web server. It includes the most widely used extensions.
  - **minimal** - This profile installs only the command line for scripting with PHP without using a web server.
  - **devel** - This profile includes packages from the common profile and additional packages for development purposes.
    For example, to install PHP 8.1 for use without a web server, use:

```
# dnf module install php:8.1/minimal
```

Additional resources

- Managing software with the DNF tool

6.2. USING THE PHP SCRIPTING LANGUAGE WITH A WEB SERVER

6.2.1. Using PHP with the Apache HTTP Server

In Red Hat Enterprise Linux 9, the **Apache HTTP Server** enables you to run PHP as a FastCGI process server. FastCGI Process Manager (FPM) is an alternative PHP FastCGI daemon that allows a website to manage high loads. PHP uses FastCGI Process Manager by default in RHEL 9.

You can run the PHP code using the FastCGI process server.

Prerequisites

- The PHP scripting language is installed on your system.

Procedure

1. Install the **httpd** package:

```
# dnf install httpd
```

2. Start the **Apache HTTP Server**:

```
# systemctl start httpd
```

   Or, if the **Apache HTTP Server** is already running on your system, restart the **httpd** service after installing PHP:

```
# systemctl restart httpd
```

3. Start the **php-fpm** service:

```
# systemctl start php-fpm
```

4. Optional: Enable both services to start at boot time:

```
# systemctl enable php-fpm httpd
```

5. To obtain information about your PHP settings, create the **index.php** file with the following content in the **/var/www/html/** directory:

```
# echo '<?php phpinfo(); ?>' > /var/www/html/index.php
```

6. To run the **index.php** file, point the browser to:

```
http://<hostname>/
```

7. Optional: Adjust configuration if you have specific requirements:

   - **/etc/httpd/conf/httpd.conf** - generic **httpd** configuration
   - **/etc/httpd/conf.d/php.conf** - PHP-specific configuration for **httpd**
   - **/usr/lib/systemd/system/httpd.service.d/php-fpm.conf** - by default, the **php-fpm** service is started with **httpd**

- **/etc/php-fpm.conf** - FPM main configuration
- **/etc/php-fpm.d/www.conf** - default **www** pool configuration

Example 6.1. Running a "Hello, World!" PHP script using the Apache HTTP Server

1. Create a **hello** directory for your project in the **/var/www/html/** directory:

```
# mkdir hello
```

2. Create a **hello.php** file in the **/var/www/html/hello/** directory with the following content:

```
# <!DOCTYPE html>
<html>
<head>
<title>Hello, World! Page</title>
</head>
<body>
<?php
   echo 'Hello, World!';
?>
</body>
</html>
```

3. Start the **Apache HTTP Server**:

```
# systemctl start httpd
```

4. To run the **hello.php** file, point the browser to:

```
http://<hostname>/hello/hello.php
```

As a result, a web page with the “Hello, World!” text is displayed.

Additional resources

- Setting up the Apache HTTP web server

6.2.2. Using PHP with the nginx web server

You can run PHP code through the **nginx** web server.

Prerequisites

- The PHP scripting language is installed on your system.

Procedure

1. Install the **nginx** package:

```
# dnf install nginx
```

2. Start the **nginx** server:

```
# systemctl start nginx
```

Or, if the **nginx** server is already running on your system, restart the **nginx** service after installing PHP:

```
# systemctl restart nginx
```

3. Start the **php-fpm** service:

```
# systemctl start php-fpm
```

4. Optional: Enable both services to start at boot time:

```
# systemctl enable php-fpm nginx
```

5. To obtain information about your PHP settings, create the **index.php** file with the following content in the **/usr/share/nginx/html/** directory:

```
# echo '<?php phpinfo(); ?>' > /usr/share/nginx/html/index.php
```

6. To run the **index.php** file, point the browser to:

```
http://<hostname>/
```

7. Optional: Adjust configuration if you have specific requirements:

- **/etc/nginx/nginx.conf** - **nginx** main configuration
- **/etc/nginx/conf.d/php-fpm.conf** - FPM configuration for **nginx**
- **/etc/php-fpm.conf** - FPM main configuration
- **/etc/php-fpm.d/www.conf** - default **www** pool configuration

Example 6.2. Running a "Hello, World!" PHP script using the nginx server

1. Create a **hello** directory for your project in the **/usr/share/nginx/html/** directory:

```
# mkdir hello
```

2. Create a **hello.php** file in the **/usr/share/nginx/html/hello/** directory with the following content:

```
# <!DOCTYPE html>
<html>
<head>
<title>Hello, World! Page</title>
</head>
<body>
<?php
    echo 'Hello, World!';
```

```
?>
</body>
</html>
```

3. Start the **nginx** server:

```
# systemctl start nginx
```

4. To run the **hello.php** file, point the browser to:

```
http://<hostname>/hello/hello.php
```

As a result, a web page with the “Hello, World!” text is displayed.

Additional resources

- Setting up and configuring NGINX

6.3. RUNNING A PHP SCRIPT USING THE COMMAND LINE

A PHP script is usually run using a web server, but also can be run using the command line.

Prerequisites

- The PHP scripting language is installed on your system.

Procedure

1. In a text editor, create a ***filename*.php** file
   Replace *filename* with the name of your file.
2. Execute the created ***filename*.php** file from the command line:

```
# php filename.php
```

Example 6.3. Running a "Hello, World!" PHP script using the command line

1. Create a **hello.php** file with the following content using a text editor:

```
<?php
    echo 'Hello, World!';
?>
```

2. Execute the **hello.php** file from the command line:

```
# php hello.php
```

As a result, “Hello, World!” is printed.

6.4. ADDITIONAL RESOURCES

- **httpd(8)** – The manual page for the **httpd** service containing the complete list of its command-line options.
- **httpd.conf(5)** – The manual page for **httpd** configuration, describing the structure and location of the **httpd** configuration files.
- **nginx(8)** – The manual page for the **nginx** web server containing the complete list of its command-line options and list of signals.
- **php-fpm(8)** – The manual page for PHP FPM describing the complete list of its command-line options and configuration files.