Oracle Linux 8

Building RPM Packages From Source

F44226-04
April 2022

Oracle Linux 8 Building RPM Packages From Source,

F44226-04

Contents

Preface

Conventions v
Documentation Accessibility v
Access to Oracle Support for Accessibility v
Diversity and Inclusion v

1 About Building Source Packages

About Source RPMS 1-1
About the Module Build Service 1-2

2 General Requirements

Enable Required Developer Repositories 2-1
Install Packaging Tools and the Module Build Service 2-1

3 Building Non-Modular Source RPM Packages

Using rpmbuild Directly 3-1
Using the mock Utility to Build Sources 3-2

4 Building Modules

Create Git Repositories for Module Sources 4-1
- Download Module Sources 4-1
- Generate a Working modulemd and Plan the Required Git Repositories 4-3
- Create Source Git Repositories and Branches 4-6
  - Remote Git Repositories 4-6
  - Local Git Repositories 4-7

Configure MBS for Remote Source Repositories 4-8
Module Build Service Mock Configuration 4-8
Triggering a Build 4-10

Test a Module Build 4-10

Preface

Oracle® Linux 8: Building RPM Packages From Source describes how to set up the Module Build Service (MBS) to build modular RPM packages from source for Oracle Linux 8. The information in this document is for training purposes and can be used in a development environment to build software sources as modular binary RPM packages for Oracle Linux 8 so that changes to code can be easily tested across test platforms. Note that Oracle does not support packages that are built externally and also does not support all of the components within a build environment.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility/.

For information about the accessibility of the Oracle Help Center, see the Oracle Accessibility Conformance Report at https://www.oracle.com/corporate/accessibility/templates/t2-11535.html.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to

build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

About Building Source Packages

This chapter provides overview information about building and packaging source packages as binary RPM packages. A distinction is made between building standard non-modular source packages and building source that is distributed within DNF modules and streams. Some information on how to obtain source packages is also provided.

> **NOT_SUPPORTED:**
>
> This documentation illustrates how source packages for Oracle Linux 8 can be built and packaged as binary RPM packages to effectively bootstrap Oracle Linux 8 from source. The information that is provided here is intended to assist developers in understanding how to build and package source code for Oracle Linux 8. Oracle does not support packages that are built externally. These instructions do not imply support for the infrastructure that is described or for any package build processes.

About Source RPMS

All of the sources for Oracle Linux packages are available on the Oracle Linux yum server and ULN. For any package, you can download the relevant source RPM and rebuild the binary equivalent by using the `rpmbuild` utility. Source RPM packages contain the source code, which is provided as a tarball. Also included are any other patches that Oracle may apply during the build, as well as a SPEC file that provides important build information, which includes a list of the build dependencies that are required to build the package and build instructions for any actions that should be taken at different stages of the package build process.

Obtaining source packages is straightforward. You can manually download each source package that you wish to build by using the `dnf download` command. For example, to download the latest source package for the UEK package that is shipped for Oracle Linux 8, run:

```
sudo dnf download --source kernel-uek
```

Note that you can encounter some issues when downloading the source RPM packages by using the `dnf download` command if the package is released as a module. If necessary, you can manually download source packages directly from https://yum.oracle.com/oracle-linux-8.html.

Modular packages contain metadata with information about the packages that are part of the module and are made available in a modular RPM repository that includes a YAML file that provides more information about the modules and streams available. You cannot download sources for modular packages directly by using the `dnf download` command. For more information about downloading modular packages, see Download Module Sources.

More information about DNF modularity is provided in Oracle® Linux: Managing Software on Oracle Linux.

If you intend to build the majority of packages directly from source, you should consider creating a mirror of the yum repositories or ULN channels where the packages are located. On an Oracle Linux 8 system, you can easily create a mirror of all of the repositories or channels that your system is subscribed to by running the following command:

```
mkdir source_rpms

sudo dnf reposync --source -p source_rpms
```

The previous command downloads all of the sources for every package in every repository for which the system is subscribed. You must ensure that the file system on which you host these source RPM packages has sufficient space to store these packages. You can find out how much space is used by each repository by running `dnf repolist -v`.

For modular packages, the `dnf reposync` command downloads the source RPM packages for each defined module stream, which means that multiple versions of the same source package may exist in the repository mirror.

More information about the RPM packaging format can be found at https://rpm.org/documentation and a detailed guide to RPM packaging is available at https://rpm-packaging-guide.github.io/.

About the Module Build Service

The Module Build Service (MBS) is a build management utility that coordinates build jobs specifically for DNF modules. This utility provides an interface to:

- manage client-side tooling,
- validate build configuration for each module,
- set up the appropriate build environment,
- and handle build scheduling, tracking, and reporting.

The upstream project is hosted at https://pagure.io/fm-orchestrator.

The Mock build tool is used within MBS to facilitate builds of the source packages. Mock builds source RPM packages within a chroot environment. This tool simply sets up the environment and populates it based on the contents of a configuration file. The source RPM packages are then built within the chroot environment and packaged as RPM binaries. The Mock build tool can be threaded within MBS to improve performance and speed up builds for many modules in parallel.

Oracle provides packages for MBS and its dependencies in a developer repository. These packages include some initial configuration that can help you get started with building modules directly from the sources that are provided by Oracle.

2
General Requirements

This chapter describes the required steps for building both modular and standard source packages. Preparing a system as a build server requires that you enable developer repositories which contain unsupported software. You must ensure that your system is current with the latest updates and you must install the required build packages and dependencies .

If you are building modular packages, several additional setup steps are required. For instance you are also required to either host source content yourself either on a remote Git service or locally within your own Git repositories. Modular package repositories must be populated with source code and the Module Build Service must be configured appropriately. Furthermore, you must configure the Module Build Service and related tools so that modular RPMs can be generated. Information on this is provided in more detail in Building Modules.

Enable Required Developer Repositories

> **Important:**
>
> Enabling the following repositories can result in a system running unsupported packages. A system that is used for the purpose of building modular RPM packages from source is, by nature, an unsupported system. If you proceed with these instructions, use a dedicated system for which you do not intend to obtain direct support from Oracle.

Many of the development tools, libraries and dependencies required to build the source available for Oracle Linux 8 are hosted in unsupported developer repositories. Before proceeding make sure that your system is up to date so that you have the most recent version of any of the release packages installed:

```
sudo dnf update
```

Install the `oracle-epel-release-el8` package if it is not already installed and enable all of the required developer repositories:

```
sudo dnf install oracle-epel-release-el8

sudo dnf config-manager --enable ol8_codeready_builder

sudo dnf config-manager --enable ol8_developer_EPEL

sudo dnf config-manager --enable ol8_distro_builder
```

Install Packaging Tools and the Module Build Service

Install the tools and utilities required to build a RPM binaries from a source tar files and a SPEC file, along with MBS:

```
sudo dnf install -y rpm-build yum-utils mock module-build-service oracle-mbs-
tools
```

You may need to install some additional dependencies before you are able to start building all of your source packages and to meet any of your own build requirements. Typically, build teams additionally install the following:

```
sudo dnf install -y gcc python3-service-identity
```

Optionally, if you use Git Large File Storage (LFS) to store RPM sources, install the `git-lfs` package:

```
sudo dnf install git-lfs
```

3
Building Non-Modular Source RPM Packages

This chapter describes how to build non-modular source RPM packages. Two methods are provided: the first is more direct and uses the `rpmbuild` utility directly; the second requires that you set up the `mock` utility and configure the system to process builds using this tool. There are several advantages to using the `mock` utility to build packages and you should consider using this tool if you intend to do mutliple or regular builds of different packages.

Using rpmbuild Directly

The simplest approach for building is to use the `rpmbuild --rebuild` command to build a binary RPM package directly from the source RPM package. This tool takes the source RPM package and extracts it into a standard hierarchy within `~/rpmbuild/`; the tool checks that all of the build dependencies specified in the associated SPEC file that is stored in `~/rpmbuild/SPECS` are satisfied; the source tarballs in `~/rpmbuild/SOURCES` are extracted into a build directory, and are each built before being packaged into a binary RPM package, which is stored in `~/rpmbuild/RPMS`.

> **Important:**
>
> Do not build source RPM packages as the `root` user. The build processes often run scripts and processes that you may not have full control over and that could easily cause system failure or could compromise a system.

> **Note:**
>
> When using the `rpmbuild` tool to build source RPM packages directly, you may see warning messages similar to the following:
>
> ```
> warning: user mockbuild does not exist - using root
> ```
>
> These warnings can be ignored, as they are related to the way these source packages are set up for build by using the `mock` utility within a more complex build environment.

If the build dependencies that are specified in the SPEC file are not satisfied, the `rpmbuild --rebuild` command errors out and lists the failed dependencies, for example:

```
rpmbuild --rebuild bash-4.4.19-12.el8.src.rpm

Installing bash-4.4.19-12.el8.src.rpm
...
error: Failed build dependencies:
  autoconf is needed by bash-4.4.19-12.el8.x86_64
```

```
ncurses-devel is needed by bash-4.4.19-12.el8.x86_64
texinfo is needed by bash-4.4.19-12.el8.x86_64
```

You can either manually install each dependency that is listed in the warning; or, you can use the `dnf builddep` command to resolve all of the build dependencies specified in the SPEC file that was extracted when you ran the `rpmbuild`command, for example:

```
sudo dnf builddep -y ~/rpmbuild/SPECS/bash.spec
```

If all of the build dependencies are satisfied and the source package has been created appropriately, the `rpmbuild --rebuild` command completes after the binary package build is complete. You can access the package in `~/rpmbuild/RPMS`.

If you need to modify source, apply alternative patches, or edit the SPEC file to perform alternate actions during different stages of the build process, you can do so and then build the binary and source packages again directly from the SPEC file, for example:

```
rpmbuild -ba ~/rpmbuild/SPECS/bash.spec
```

Using the mock Utility to Build Sources

The `mock` utility provides a wrapper that can help build sources safely and can handle build dependency resolution for you in the background. This tool sets up a chroot directory to handle the build process. By creating a chroot environment, several potential build issues are handled automatically. Most importantly, the build process itself is kept safe from affecting the host system. The chroot environment can install build dependency packages and everything that is required to complete the build, without actually installing unwanted packages into the host system so that the host system can remain relatively clean of build artifacts. Because the chroot environment contains the build process and protects the host system, build processes are effectively privileged within the contained environment, which helps facilitate a straightforward build and minimize potential errors. Finally, build dependencies can be automatically resolved by using the `mock` utility, which can safely and automatically trigger the appropriate `dnf builddep` command, as required.

The `mock` utility requires some preliminary configuration before using it to build for Oracle Linux sources. A useful starting point is to create a `/etc/mock/templates/ol-8.tpl` template file with a configuration similar to the following:

```
config_opts['chroot_setup_cmd'] = 'install tar gcc-c++ redhat-rpm-config
oraclelinux-release which xz sed \
                                  make bzip2 gzip gcc coreutils unzip shadow-
utils diffutils cpio bash gawk \
                                  rpm-build info patch util-linux findutils
grep'
config_opts['dist'] = 'el8'  # only useful for --resultdir variable subst
config_opts['extra_chroot_dirs'] = [ '/run/lock', ]
config_opts['releasever'] = '8'
config_opts['package_manager'] = 'dnf'
config_opts['root'] = 'ol-8-{{ target_arch }}'

config_opts['dnf.conf'] = """
[main]
keepcache=1
debuglevel=2
reposdir=/dev/null
```

```
logfile=/var/log/yum.log
retries=20
obsoletes=1
gpgcheck=1
assumeyes=1
syslog_ident=mock
syslog_device=
install_weak_deps=0
metadata_expire=0
best=1
module_platform_id=platform:el8
protected_packages=

# repos
[ol8_baseos_latest]
name=Oracle Linux 8 BaseOS Latest ($basearch)
baseurl=https://yum.oracle.com/repo/OracleLinux/OL8/baseos/latest/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_appstream]
name=Oracle Linux 8 Application Stream ($basearch)
baseurl=https://yum.oracle.com/repo/OracleLinux/OL8/appstream/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_codeready_builder]
name=Oracle Linux 8 CodeReady Builder ($basearch) - Unsupported
baseurl=https://yum.oracle.com/repo/OracleLinux/OL8/codeready/builder/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_distro_builder]
name=Oracle Linux 8 Distro Builder ($basearch) - Unsupported
baseurl=https://yum.oracle.com/repo/OracleLinux/OL8/distro/builder/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_developer_EPEL]
name=Oracle Linux $releasever EPEL Packages for Development ($basearch)
baseurl=https://yum.oracle.com/repo/OracleLinux/OL8/developer/EPEL/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

"""
```

Replace `/etc/mock/default.cfg` with the following content, substituting `x86_64` with `aarch64` if you intend to build packages for Arm platforms:

```
include('templates/ol-8.tpl')
config_opts['target_arch'] = 'x86_64'
config_opts['legal_host_arches'] = ('x86_64',)
```

Before any user can use the `mock` utility, the user must be added to the `mock` group. You can do this by running the following command:

```
sudo /usr/sbin/usermod -a -G mock $USER
```

Note that if the user is currently logged in, the user may need to log out and then log back in to the system for the change in group permissions to become active.

When the configuration is in place, you can start to build source packages by using the `mock` utility. For example, to simply build a package directly from a source RPM package:

```
mock --rebuild bash-4.4.19-12.el8.src.rpm
```

Packages are built in a chroot environment that is created for the build root that is defined in the template. You can access the file system used in the chroot in `/var/lib/mock`. For example, using the configurations presented in this documentation would be similar to the following:

```
ls /var/lib/mock/ol-8-x86_64/root/builddir/build/SRPMS/

bash-4.4.19-12.el8.src.rpm

ls -lh /var/lib/mock/ol-8-x86_64/root/builddir/build/RPMS/

total 4.9M
-rw-r--r--. 1 root mock 1.6M Feb 3 02:10 bash-4.4.19-12.el8.x86_64.rpm
-rw-r--r--. 1 root mock 1.2M Feb 3 02:10 bash-debuginfo-4.4.19-12.el8.x86_64.rpm
-rw-r--r--. 1 root mock 841K Feb 3 02:10 bash-
debugsource-4.4.19-12.el8.x86_64.rpm
-rw-r--r--. 1 root mock 113K Feb 3 02:10 bash-devel-4.4.19-12.el8.x86_64.rpm
-rw-r--r--. 1 root mock 1.3M Feb 3 02:10 bash-doc-4.4.19-12.el8.x86_64.rpm
```

If you need to work with build artifacts within the mock chroot, you can also do the following:

```
mock --shell

INFO: mock.py version 2.4 starting (python version = 3.6.8)...
Start(bootstrap): init plugins
INFO: selinux enabled
Finish(bootstrap): init plugins
Start: init plugins
INFO: selinux enabled
Finish: init plugins
INFO: Signal handler active
Start: run
Start(bootstrap): chroot init
INFO: calling preinit hooks
INFO: enabled root cache
INFO: enabled package manager cache
Start(bootstrap): cleaning package manager metadata
Finish(bootstrap): cleaning package manager metadata
INFO: enabled HW Info plugin
Finish(bootstrap): chroot init
Start: chroot init
INFO: calling preinit hooks
INFO: enabled root cache
INFO: enabled package manager cache
Start: cleaning package manager metadata
Finish: cleaning package manager metadata
INFO: enabled HW Info plugin
Finish: chroot init
Start: shell
```

```
ls -lah /builddir/

total 20K
drwx------. 1 root 1000 120 Feb  4 09:21 .
dr-xr-xr-x. 1 root root 212 Feb  3 02:06 ..
-rw-------. 1 root root  50 Feb  4 09:21 .bash_history
-rw-r--r--. 1 root 1000  18 Aug  2  2020 .bash_logout
-rw-r--r--. 1 root 1000 141 Aug  2  2020 .bash_profile
-rw-r--r--. 1 root 1000 376 Aug  2  2020 .bashrc
-rw-rw-r--. 1 root root 130 Feb  3 02:06 .rpmmacros
drwxrwxr-x. 1 root 1000  88 Feb  3 02:06 build
```

You can find out more about mock in the MOCK(1) manual page or at https://github.com/rpm-software-management/mock/.

4
Building Modules

This chapter describes how to build DNF modules. Some initial setup is required and this process is more complicated than building individual source RPMs as a module can be comprised of many source components and dependencies. The steps described in this chapter provide information on how to get set up and started building module from the source packages provided for Oracle Linux 8, using the Module Build Service.

Create Git Repositories for Module Sources

MBS pulls sources from a structured Git repository and uses a module definition file to resolve exactly which source packages are required to build a particular module stream and version.

To facilitate this requirement, you must do the following:

- Set up Git repositories for each module that you intend to build
- Create branches within each repository for the different streams within the module
- Populate the branches with the source code that is required to build the module packages

Download Module Sources

Module sources can be downloaded from the Oracle Linux yum server; although, this task does require additional analysis to determine which source packages are required.

You can list all of the modules that are available within the yum repositories that you are subscribed to by running the `dnf module list` command. You can check the information for any module and stream by using the `dnf module info` *`module_name:stream`* command. To limit the information that is returned to a specific module stream, version, context, architecture, and platform, use the full module reference. For example, you would run the following command to view the information for version `8030020200818000036` of the `2.4` stream of the `httpd` module:

```
dnf module info httpd:2.4:8030020200818000036

Name : httpd
Stream : 2.4 [d][a]
Version : 8030020200818000036
Context : 9edba152
Architecture : x86_64
Profiles : common [d], devel, minimal
Default profiles : common
Repo : ol8_appstream
Summary : Apache HTTP Server
Description : Apache httpd is a powerful, efficient, and extensible HTTP server.
Requires : platform:[el8]
Artifacts : httpd-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: httpd-devel-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: httpd-filesystem-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.noarch
```

```
: httpd-manual-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.noarch
: httpd-tools-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: mod_http2-0:1.15.7-2.module+el8.3.0+7816+49791cfd.x86_64
: mod_ldap-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: mod_md-1:2.0.8-8.module+el8.3.0+7816+49791cfd.x86_64
: mod_proxy_html-1:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: mod_session-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
: mod_ssl-1:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64

Hint: [d]efault, [e]nabled, [x]disabled, [i]nstalled, [a]ctive
```

Note that the `Artifacts` section of the output provides the list of packages that are available for the module.

A single source package can be used to build several binary packages. To determine which source packages are used for each binary package that is listed, you must use the `dnf repoquery` command and disable modular filtering, for example:

```
dnf repoquery httpd-
filesystem-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.noarch --source -q --
disable-modular-filtering

httpd-2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.src.rpm
```

Repeat the previous query for each binary package listed in the `Artifacts` section of the information that is returned from the yum server for the module.

When you have a list of all of the source packages to be downloaded, you can query the yum server for the download URL to use to download the source package, for example:

```
dnf repoquery httpd-2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.src --location -q
--disable-modular-filtering

https://yum.oracle.com/repo/OracleLinux/OL8/appstream/x86_64/getPackageSource/
httpd-2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.src.rpm
```

Use the URL that is returned by the command to download the source package file. For example, use the `curl` command to download the package to the current directory:

```
curl -O https://yum.oracle.com/repo/OracleLinux/OL8/appstream/x86_64/
getPackageSource/httpd-2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.src.rpm
```

Repeat these steps to download each source package from the Oracle Linux yum server.

Tip:

If a module for which you intend to build the sources is enabled, you can automate many of these steps in a simple bash loop, as shown in the following example:

```
for bin in $(dnf module repoquery nginx:1.14 -q);
 do
 echo "Checking for sources in $bin"
 for src in $(dnf repoquery $bin --source -q);
  do
   echo "-- Checking source package $src"
   if [ ! -f $src ]
     then
      URL=$(dnf repoquery $(echo "${src%.*}") -q --location --disable-modular-
filtering)
      echo "----> Downloading $URL"
      curl -O $URL;
   fi
  done
done
```

It is not always desirable to enable modules on the build system and enabling some modules could cause unintended conflicts. This tip is provided for users who are more familiar with their build environment and what they intend to build.

Generate a Working modulemd and Plan the Required Git Repositories

Module configuration is defined in a YAML-formatted `modulemd` document that contains all of the metadata required to package and build a module. More information about the contents and format of the modulemd document is available at https://docs.fedoraproject.org/en-US/modularity/policies/packaging-guidelines/.

MBS uses the module metadata configuration to determine which Git repository and branch to use to access the source files that are used to build packages for the module.

You can use the `dnf module info` command with the verbose switch to get the modulemd information that was used to build a module, for example:

```
dnf module info httpd:2.4:8030020200818000036 -v

Loaded plugins: builddep, changelog, config-manager, copr, debug, debuginfo-install,
download, generate_completion_cache,
  needs-restarting, playground, repoclosure, repodiff, repograph, repomanage, reposync
DNF version: 4.2.23
cachedir: /var/cache/dnf
User-Agent: constructed: 'libdnf (Oracle Linux Server 8.3; server; Linux.x86_64)'
repo: using cache for: ol8_developer_EPEL
ol8_developer_EPEL: using metadata from Fri 19 Feb 2021 09:56:59 PST.
repo: using cache for: ol8_baseos_latest
ol8_baseos_latest: using metadata from Tue 23 Feb 2021 11:19:55 PST.
repo: using cache for: ol8_appstream
ol8_appstream: using metadata from Tue 23 Feb 2021 11:49:55 PST.
repo: using cache for: ol8_codeready_builder
ol8_codeready_builder: using metadata from Tue 23 Feb 2021 11:50:47 PST.
repo: using cache for: ol8_distro_builder
ol8_distro_builder: using metadata from Tue 26 Jan 2021 14:36:37 PST.
repo: using cache for: ol8_UEKR6
```

```
ol8_UEKR6: using metadata from Tue 16 Feb 2021 09:29:11 PST.
Last metadata expiration check: 2:28:53 ago on Wed 24 Feb 2021 04:25:21 PST.
Completion plugin: Generating completion cache...
---
document: modulemd
version: 2
data:
  name: httpd
  stream: 2.4
  version: 8030020200818000036
  context: 9edba152
  arch: x86_64
  summary: Apache HTTP Server
  description: >-
    Apache httpd is a powerful, efficient, and extensible HTTP server.
  license:
    module:
    - MIT
    content:
    - ASL 2.0
  xmd:
    mbs:
      buildrequires:
        platform:
          context: 32e30060
          filtered_rpms: []
          ref:
          stream: el8
          version: 20190214123456
      commit:
      mse: TRUE
      rpms:
        httpd:
          ref: 36bae5ca8c2cfed909cbf9bb0d7d5100ae849344
        mod_http2:
          ref: 277d39ae32712ce196bf1dab8dbcc4e636cc0a44
        mod_md:
          ref: fe6eebe9285d77b75baa3da7c313fb16682eaf46
      scmurl:
  dependencies:
  - buildrequires:
      platform: [el8]
    requires:
      platform: [el8]
  references:
    documentation: https://httpd.apache.org/docs/2.4/
    tracker: https://bz.apache.org/bugzilla/
  profiles:
    common:
      rpms:
      - httpd
      - httpd-filesystem
      - httpd-tools
      - mod_http2
      - mod_ssl
    devel:
      rpms:
      - httpd
      - httpd-devel
      - httpd-filesystem
      - httpd-tools
```

```
    minimal:
      rpms:
      - httpd
  api:
    rpms:
    - httpd
    - httpd-devel
    - httpd-filesystem
    - mod_ssl
  components:
    rpms:
      httpd:
        rationale: Apache httpd
        ref: stream-2.4-rhel-8.3.0
        buildorder: 10
        arches: [aarch64, i686, x86_64]
      mod_http2:
        rationale: HTTP/2 support for Apache httpd
        ref: stream-2.4-rhel-8.3.0
        buildorder: 20
        arches: [aarch64, i686, x86_64]
      mod_md:
        rationale: Certificate provisioning using ACME for Apache httpd
        ref: stream-2.4-rhel-8.3.0
        buildorder: 20
        arches: [aarch64, i686, x86_64]
  artifacts:
    rpms:
    - httpd-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - httpd-devel-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - httpd-filesystem-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.noarch
    - httpd-manual-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.noarch
    - httpd-tools-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - mod_http2-0:1.15.7-2.module+el8.3.0+7816+49791cfd.x86_64
    - mod_ldap-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - mod_md-1:2.0.8-8.module+el8.3.0+7816+49791cfd.x86_64
    - mod_proxy_html-1:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - mod_session-0:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
    - mod_ssl-1:2.4.37-30.0.1.module+el8.3.0+7816+49791cfd.x86_64
...
```

Note that the modulemd content that is provided by this output contains additional build artifacts and cannot be used directly by MBS without modification. To make this content usable, several keys in the YAML content must be removed, including the `xmd` and `artifacts` entries. Note that the YAML content only starts in the output provided by `dnf module info` at the line containing: `---`.

You can either save the output from the `dnf module info` query and edit it manually to generate a working modulemd file; or you can save the output from this command and run it through the `convert_repodata_modulemd.py` script, include in the `oracle-mbs-tools` package, to automatically parse it and generate a working YAML modulemd build configuration file. For example:

```
dnf module info httpd:2.4:8030020200818000036 -v | convert_repodata_modulemd.py -i

ls *.yaml

httpd-2.4-8030020200818000036-9edba152.yaml
```

Note that you may need to change the version number in the modulemd configuration file, as you cannot build an existing version of the same module. The typical version label for modules uses the following convention:

<distribution version><update level><optional module version><date and time>

So for a module from Oracle Linux 8.3 with a version 01 built on 12 April 2021 at 15:30:22, the version would be: 8030120210412153022.

You should always check that the modulemd YAML file is valid before using it. You can check validity using the `modulemd-validator` command. For example, run:

```
modulemd-validator httpd-2.4-8030020200818000036-9edba152.yaml
```

The following information in the modulemd YAML file is worth noting, as it describes how source is pulled from Git, as well as the steps you might need to take before you are able to build a module successfully:

- `dependencies`/`buildrequires`: before you are able to build the module, all build dependencies must be met, specifically any modules listed under the `buildrequires` entry must be built and made available before you can proceed with a local build. If you are only interested in building a particular module, these build dependencies can be met at build time and MBS can pull the required modules from your configured yum repositories, as required.
- `components`/`rpms`: each component RPM name is used to define the Git repository name where the source is located at the base URL that MBS is configured to use. The `ref` parameter describes which branch should be used to build the correct RPM packages for the particular module stream defined in the modulemd file.

Search the modulemd YAML file for the `components:rpms` entries. Each component RPM entry represents a Git repository and is related to a corresponding source package. The `ref` entry provided for each RPM is used to define the branch name within the Git repository. For example, from the output provided above, the `httpd:2.4:8030020200818000036` module uses sources from the `stream-2.4-rhel-8.3.0` branches in each of the following three Git repositories: `httpd`, `mod_http2` and `mod_md`.

Use this information to plan corresponding Git repositories and branches, as required by the modulemd configuration information for each module and stream that you intend to build.

Create Source Git Repositories and Branches

MBS uses Git to extract the sources that it uses. Git repositories can be located either remotely or locally, but in either case, Git must be set up so that you can store sources for MBS to use. Instructions are provided here for both approaches. You must create a separate Git repository for each source component that you have downloaded. Repeat the steps provided for each source RPM.

Remote Git Repositories

If you use a remote Git service, you must use the tools that are provided by your service provider to create matching repositories to host the component source code for each source RPM that you have downloaded. The repository name must match the name of the component source RPM.

Import the sources into your remote Git repositories:

1. Clone your Git repository with the same name as the source component RPM defined in the modulemd:

   ```
   git clone git@git_server_URL:repository path/component.git

   cd component
   ```

2. Check out the branch matching the stream ref entry for the component RPM in the modulemd:

   ```
   git checkout ref
   ```

   If the branch does not yet exist, you can create a local branch with:

   ```
   git checkout -b ref
   ```

3. Extract the source package into the working directory:

   ```
   rpm2cpio /path/to/component.src.rpm | cpio -di
   ```

4. Add the sources to Git and push them to the remote server:

   ```
   git add *

   git commit -m "Import sources for component"

   git push
   ```

   If you had to create a new local branch to work in, when you push the sources to the remote server you must ensure that it stores the source in a matching branch:

   ```
   git push --set-upstream origin ref
   ```

Local Git Repositories

If you choose to host sources on localized Git repositories some reconfiguration is required within the modulemd files for each module.

1. Extract the contents of the source package into an empty directory with the same name as the source component RPM that is defined in the modulemd:

   ```
   $ mkdir component

   $ cd component

   rpm2cpio /path/to/component.src.rpm | cpio -di
   ```

2. Initialize a local Git repository

   ```
   git init
   ```

3. Add files and commit

   ```
   git add *

   git commit -m "Import sources for component"
   ```

4. Modify the modulemd file for this module to replace the `ref` entries to point to the path of the new local repository that you have created. For example modify the following entry:

   ```
   components:
       rpms:
         httpd:
   ```

```
        rationale: Apache httpd
        ref: stream-2.4-rhel-8.3.0
        buildorder: 10
        arches: [aarch64, i686, x86_64]
```

so that it reads:

```
components:
    rpms:
      httpd:
        rationale: Apache httpd
        repository: file:///home/build/httpd
        buildorder: 10
        arches: [aarch64, i686, x86_64]
```

Ensure that the repository entry provides the file path to the correct location of the repository that you have created for this source component RPM.

You must edit every `ref` entry within the modulemd to match the local repository sources that for the module that you are building.

Configure MBS for Remote Source Repositories

MBS automatically pulls sources from Git repositories and analyzes the modulemd definition document to determine the repository and branch that should be used to build a specific version of the source code. See Generate a Working modulemd and Plan the Required Git Repositories for more information. If you have chosen to use a remotely hosted Git service to host your sources, you must configure MBS for the base URL where your repositories are hosted. The base URL is defined using the `RPMS_DEFAULT_REPOSITORY` parameter in the `/etc/module-build-service/config.py` file in the `BaseConfiguration` class. The default URL that is provided in the package is set to *`https://exampledomain/default-rpm-repositories/`*. Replace the default value with the location of a valid Git service where the source for your modular RPM packages can be accessed. If you are hosting your Git repositories locally on the same server and you have edited your modulemd files for this, you do not have to edit this parameter.

Because MBS uses source control management facilities like Git to access source, an additional parameter, `DISTGITS`, is set to provide the commands that are used to access and download the source into the package buildroot. The default value provided in the configuration should work correctly with the instructions that are provided in this document.

To configure MBS to use a remote Git repository , edit the `/etc/module-build-service/config.py` file and modify the following lines in the configuration under the `BaseConfiguration` class.

```
RPMS_DEFAULT_REPOSITORY = 'https://exampledomain/default-rpm-repositories/'
```

Module Build Service Mock Configuration

Before you begin building packages by using MBS, you must check and update the configuration for the Mock build tool that is used by MBS to build source packages into their binary equivalents. The global configuration file that is used by Mock when it is triggered from within MBS is located in `/etc/module-build-service/mock.cfg`.

Notably, this configuration sets the configuration for the different yum repositories that are used during the build process to resolve any build dependencies or build requirements. Since the build runs within a chroot environment, this yum configuration is separated from the host system where MBS is running.

A working example configuration is provided within the package and enables commonly required yum repositories. You may want to edit this file for additional repositories, if required.

The following contents of the configuration file are provided for reference purposes:

```
config_opts['root'] = '$root'
config_opts['target_arch'] = '$arch'
config_opts['legal_host_arches'] = ('$arch',)
config_opts['chroot_setup_cmd'] = 'install oraclelinux-release bash bzip2 coreutils
cpio diffutils findutils gawk gcc\
  gcc-c++ grep gzip info make patch redhat-rpm-config rpm-build sed yum shadow-utils
tar unzip util-linux\
  which xz $group'
config_opts['rpmbuild_networking'] = True
config_opts['use_host_resolv'] = True
config_opts['use_nspawn'] = False
config_opts['dist'] = 'el8'
config_opts['dnf_vars'] = $dnf_vars
config_opts['releasever'] = '$releasever'
config_opts['module_enable'] = $enabled_modules
config_opts['use_bootstrap_container'] = False

config_opts['yum.conf'] = """

$yum_conf

[ol8_baseos_latest]
name=Oracle Linux 8 BaseOS Latest ($basearch)
baseurl=https://yum$ociregion.oracle.com/repo/OracleLinux/OL8/baseos/latest/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_appstream]
name=Oracle Linux 8 Application Stream ($basearch)
baseurl=https://yum$ociregion.oracle.com/repo/OracleLinux/OL8/appstream/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_codeready_builder]
name=Oracle Linux 8 CodeReady Builder ($basearch) - Unsupported
baseurl=https://yum$ociregion.oracle.com/repo/OracleLinux/OL8/codeready/
builder/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_distro_builder]
name=Oracle Linux 8 Distro Builder ($basearch) - Unsupported
baseurl=https://yum$ociregion.oracle.com/repo/OracleLinux/OL8/distro/builder/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

[ol8_developer_EPEL]
```

```
name=Oracle Linux $releasever EPEL Packages for Development ($basearch)
baseurl=https://yum$ociregion.oracle.com/repo/OracleLinux/OL8/developer/
EPEL/$basearch/
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1

"""
```

Triggering a Build

A modular build is triggered by using the `mbs-manager build_module_locally` command. The command must be specified with several command-line options that describe how the build process should function. The following example illustrates a build of the packages for the `389-ds:1.4` module and stream:

```
mbs-manager build_module_locally --offline --stream 1.4 \
  --add-local-build nodejs:10:20210206155331 --file 389-ds-stream-1.4.yaml
```

The command-line switches in the example are described as follows:

- `--offline`: this option is required and uses build package requirements provided locally on the build system so that MBS is able to build in offline mode
- `--stream`: this option specifies the stream of the module that you are building for. Oracle provided module definition files specify the stream within each modulenmd file, but you should specify the stream when running the command as the value used in this option is used during the build process for file path naming.
- `--add-local-build`: this option is only required if you need to reference a local build requirement as defined in the modulemd YAML file. You specify the build requirements in the format *`module_name`*`:`*`stream`*`:`*`version`*. If you omit this option, MBS pulls the required module dependency from your configured yum repositories. If you specify a local build for any build requirements, the build must exist within `~/modulebuild/builds/`.
- `--file`: the path to the modulemd YAML file for the module that you are building.

Build logs and the final built packages are located at `~/modulebuild/builds/module-`*`module_name-stream-version`*`/results/`.

Test a Module Build

Each module build within `~/modulebuild/builds/module-`*`module_name-stream-version`*`/results/` is structured in such a way as to provide a local yum repository where you are able to access the packages for the module. As such, to test a module build, you can configure a local yum repository and then test that the you can use the `dnf` command to query the module.

For example, create a yum repository config at `/etc/yum.repos.d/local.repo` similar to the following:

```
[local_mbs_repo]
name=Local MBS Repository
baseurl=file:///home/user/modulebuild/builds/module-module_name-stream-timestamp/
results/
gpgcheck=0
enabled=1
```

Per the example, you would replace the *user*, *module_name*, *stream* and *timestamp* variables with values that are appropriate to the environment you are testing.

Run the following `dnf` commands to validate that the module information is correct and that the packages for the module stream can be installed:

```
sudo dnf module list
```

```
sudo dnf module install module_name:stream
```