Red Hat Enterprise Linux 9.0

Customizing Anaconda

Changing the installer appearance and creating custom add-ons on Red Hat Enterprise Linux

Last Updated: 2024-06-25

Red Hat Enterprise Linux 9.0 Customizing Anaconda

Changing the installer appearance and creating custom add-ons on Red Hat Enterprise Linux

Legal Notice

Abstract

Anaconda is the installer used by Red Hat Enterprise Linux. You can customize Anaconda for extending capabilities when you install RHEL in your environment.

Table of Contents

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 ANACONDA CUSTOMIZATION

1.1. INTRODUCTION TO ANACONDA CUSTOMIZATION

The Red Hat Enterprise Linux and Fedora installation program, **Anaconda**, brings many improvements in its most recent versions. One of these improvements is enhanced customizability. You can now write add-ons to extend the base installer functionality, and change the appearance of the graphical user interface.

This document will explain how to customize the following:

- Boot menu - pre-configured options, color scheme and background
- Appearance of the graphical interface - logo, backgrounds, product name
- Installer functionality - add-ons which can enhance the installer by adding new Kickstart commands and new screens in the graphical and textual user interfaces

Also note that this document applies only to Red Hat Enterprise Linux 8 and Fedora 17 and later.

IMPORTANT

Procedures described in this book are written for Red Hat Enterprise Linux 9 or a similar system. On other systems, the tools and applications used (such as **genisoimage** for creating custom ISO images) may be different, and procedures may need to be adjusted.

CHAPTER 2. PERFORMING THE PRE-CUSTOMIZATION TASKS

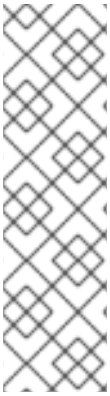
2.1. WORKING WITH ISO IMAGES

In this section, you will learn how to:

- Extract a Red Hat ISO.
- Create a new boot image containing your customizations.

2.2. DOWNLOADING RH BOOT IMAGES

Before you begin to customize the installer, download the Red Hat-provided boot images. You can obtain Red Hat Enterprise Linux 9 boot media from the Red Hat Customer Portal after login to your account.



NOTE

- Your account must have sufficient entitlements to download Red Hat Enterprise Linux 9 images.
- You must download either the **Binary DVD** or **Boot ISO** image and can use any of the image variants (Server or ComputeNode).
- You cannot customize the installer using the other available downloads, such as the KVM Guest Image or Supplementary DVD; other available downloads, such as the **KVM Guest Image** or **Supplementary DVD**.

For more information about the Binary DVD and Boot ISO downloads, see Red Hat Enterprise Linux 9 Performing an advanced RHEL 9 installation.

2.3. EXTRACTING RED HAT ENTERPRISE LINUX BOOT IMAGES

Perform the following procedure to extract the contents of a boot image.

Procedure

1. Ensure that the directory ***/mnt/iso*** exists and nothing is currently mounted there.
2. Mount the downloaded image.

```
# mount -t iso9660 -o loop path/to/image.iso /mnt/iso
```

   Where *path/to/image.iso* is the path to the downloaded boot image.

3. Create a working directory where you want to place the contents of the ISO image.

```
$ mkdir /tmp/ISO
```

4. Copy all contents of the mounted image to your new working directory. Make sure to use the **-p** option to preserve file and directory permissions and ownership.

```
# cp -pRf /mnt/iso /tmp/ISO
```

5. Unmount the image.

```
# umount /mnt/iso
```

Additional resources

- For detailed download instructions and description of the Binary DVD and Boot ISO downloads, see the Red Hat Enterprise Linux 9 .

CHAPTER 3. CUSTOMIZING THE BOOT MENU

This section provides information about what the Boot menu customization is, and how to customize it.

Prerequisites:

For information about downloading and extracting Boot images, see Extracting Red Hat Enterprise Linux boot images

The Boot menu customization involves the following high-level tasks:

1. Complete the prerequisites.
2. Customize the Boot menu.
3. Create a custom Boot image.

3.1. CUSTOMIZING THE BOOT MENU

The *Boot menu* is the menu which appears after you boot your system using an installation image. Normally, this menu allows you to choose between options such as **Install Red Hat Enterprise Linux**, **Boot from local drive** or **Rescue an installed system**. To customize the Boot menu, you can:

- Customize the default options.
- Add more options.
- Change the visual style (color and background).

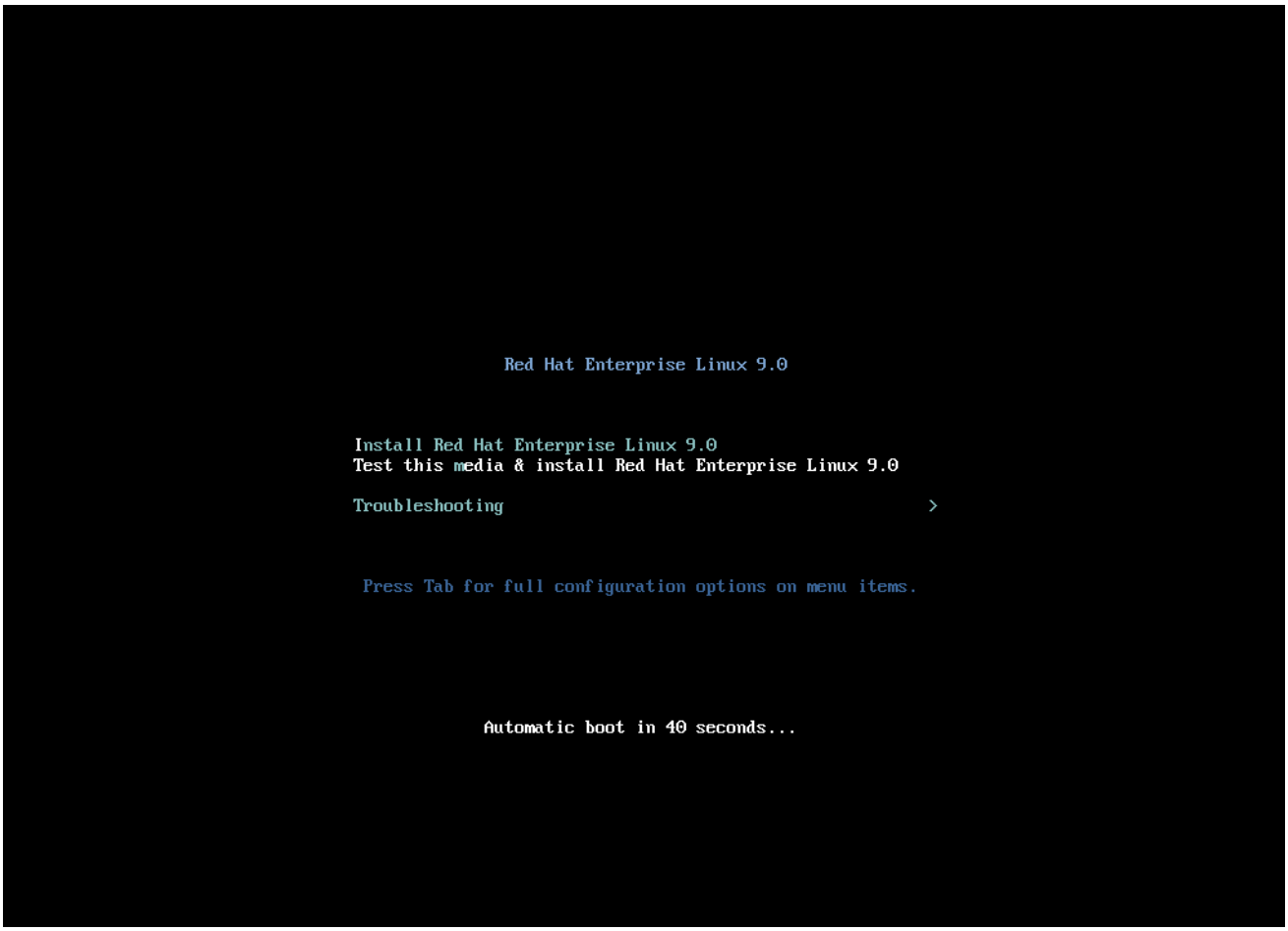
An installation media consists of **ISOLINUX** and **GRUB2** boot loaders. The **ISOLINUX** boot loader is used on systems with BIOS firmware, and the **GRUB2** boot loader is used on systems with UEFI firmware. Both the boot loaders are present on all Red Hat images for AMD64 and Intel 64 systems.

Customizing the boot menu options can especially be useful with Kickstart. Kickstart files must be provided to the installer before the installation begins. Normally, this is done by manually editing one of the existing boot options to add the **inst.ks=** boot option. You can add this option to one of the pre-configured entries, if you edit boot loader configuration files on the media.

3.2. SYSTEMS WITH BIOS FIRMWARE

The **ISOLINUX** boot loader is used on systems with BIOS firmware.

Figure 3.1. ISOLINUX Boot Menu



The ***isolinux/isolinux.cfg*** configuration file on the boot media contains directives for setting the color scheme and the menu structure (entries and submenus).

In the configuration file, the default menu entry for Red Hat Enterprise Linux, **Test this media & Install Red Hat Enterprise Linux 9**, is defined in the following block:

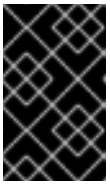
```
label check
  menu label Test this ^media & install Red Hat Enterprise Linux 9.
  menu default
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64 rd.live.check
quiet
```

Where:

- **menu label** - determines how the entry will be named in the menu. The **^** character determines its keyboard shortcut (the **m** key).
- **menu default** - provides a default selection, even though it is not the first option in the list.
- **kernel** - loads the installer kernel. In most cases it should not be changed.
- **append** - contains additional kernel options. The **initrd=** and **inst.stage2** options are mandatory; you can add others.
  For information about the options that are applicable to **Anaconda** refer to Red Hat Enterprise Linux 9 Performing a standard RHEL 9 installation Guide.

One of the notable options is **inst.ks=**, which allows you to specify a location of a Kickstart file. You can place a Kickstart file on the boot ISO image and use the inst.ks= option to specify its location; for example, you can place a **kickstart.ks** file into the image's root directory and use **inst.ks=hd:LABEL=RHEL-9-BaseOS-x86_64:/kickstart.ks**.

You can also use **dracut** options which are listed on the **dracut.cmdline(7)** man page.



IMPORTANT

When using a disk label to refer to a certain drive (as seen in the **inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64** option above), replace all spaces with **\x20**.

Other important options which are not included in the menu entry definition are:

- **timeout** - determines the time for which the boot menu is displayed before the default menu entry is automatically used. The default value is **600**, which means the menu is displayed for 60 seconds. Setting this value to **0** disables the timeout option.

NOTE

Setting the timeout to a low value such as **1** is useful when performing a headless installation. This helps to avoid the default timeout to finish.

- **menu begin** and **menu end** - determines a start and end of a *submenu* block, allowing you to add additional options such as troubleshooting and grouping them in a submenu. A simple submenu with two options (one to continue and one to go back to the main menu) looks similar to the following:

```
menu begin ^Troubleshooting
  menu title Troubleshooting

label rescue
  menu label ^Rescue a Red Hat Enterprise Linux system
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64 rescue quiet

menu separator

label returntomain
  menu label Return to ^main menu
  menu exit

menu end
```

The submenu entry definitions are similar to normal menu entries, but grouped between **menu begin** and **menu end** statements. The **menu exit** line in the second option exits the submenu and returns to the main menu.

- **menu background** - the menu background can either be a solid color (see **menu color** below), or an image in a PNG, JPEG or LSS16 format. When using an image, make sure that its dimensions correspond to the resolution set using the **set resolution** statement. Default dimensions are 640x480.

- **menu color** - determines the color of a menu element. The full format is:

> **menu color *element ansi foreground background shadow***

Most important parts of this command are:

- *element* - determines which element the color will apply to.
- *foreground* and *background* - determine the actual colors.
  The colors are described using an ***#AARRGGBB*** notation in hexadecimal format determines opacity:
- **00** for fully transparent.
- **ff** for fully opaque.
- **menu help *textfile*** - creates a menu entry which, when selected, displays a help text file.

Additional resources

- For a complete list of **ISOLINUX** configuration file options, see the Syslinux Wiki.

3.3. SYSTEMS WITH UEFI FIRMWARE

The **GRUB2** boot loader is used on systems with UEFI firmware.

The **EFI/BOOT/grub.cfg** configuration file on the boot media contains a list of preconfigured menu entries and other directives which controls the appearance and the Boot menu functionality.

In the configuration file, the default menu entry for Red Hat Enterprise Linux (**Test this media & install Red Hat Enterprise Linux 9**) is defined in the following block:

```
menuentry 'Test this media & install Red Hat Enterprise Linux 9' --class fedora --class gnu-linux --class gnu --class os {
    linuxefi /images/pxeboot/vmlinuz inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64 rd.live.check quiet
    initrdefi /images/pxeboot/initrd.img
}
```

Where:

- **menuentry** - Defines the title of the entry. It is specified in single or double quotes (**'** or **"**). You can use the **--class** option to group menu entries into different *classes*, which can then be styled differently using **GRUB2** themes.

  NOTE

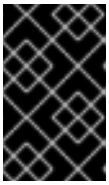
  As shown in the above example, you must enclose each menu entry definition in curly braces (**{}**).

- **linuxefi** - Defines the kernel that boots (**/images/pxeboot/vmlinuz** in the above example) and the other additional options, if any.
  You can customize these options to change the behavior of the boot entry. For details about the options that are applicable to **Anaconda**, see *Performing an advanced RHEL 9 installation* .

  One of the notable options is **inst.ks=**, which allows you to specify a location of a Kickstart file.

You can place a Kickstart file on the boot ISO image and use the inst.ks= option to specify its location; for example, you can place a **kickstart.ks** file into the image's root directory and use **inst.ks=hd:LABEL=RHEL-9-BaseOS-x86_64:/kickstart.ks**.

You can also use **dracut** options which are listed on the **dracut.cmdline(7)** man page.



IMPORTANT

When using a disk label to refer to a certain drive (as seen in the **inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64** option above), replace all spaces with **\x20**.

- **initrdefi** - location of the initial RAM disk (initrd) image to be loaded.

Other options used in the **grub.cfg** configuration file are:

- **set timeout** - determines how long is the boot menu displayed before the default menu entry is automatically used. The default value is **60**, which means the menu is displayed for 60 seconds. Setting this value to **-1** disables the timeout completely.

NOTE

Setting the timeout to **0** is useful when performing a headless installation, because this setting immediately activates the default boot entry.

- **submenu** - A *submenu* block allows you to create a sub-menu and group some entries under it, instead of displaying them in the main menu. The **Troubleshooting** submenu in the default configuration contains entries for rescuing an existing system.
The title of the entry is in single or double quotes (**'** or **"**).

  The **submenu** block contains one or more **menuentry** definitions as described above, and the entire block is enclosed in curly braces (**{}**). For example:

```
submenu 'Submenu title' {
  menuentry 'Submenu option 1' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64 xdriver=vesa
nomodeset quiet
    initrdefi /images/pxeboot/initrd.img
  }
  menuentry 'Submenu option 2' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-9-BaseOS-x86_64 rescue quiet
    initrdefi /images/initrd.img
  }
}
```

- **set default** - Determines the default entry. The entry numbers start from **0**. If you want to make the *third* entry the default one, use **set default=2** and so on.

- **theme** - determines the directory which contains **GRUB2** theme files. You can use the themes to customize visual aspects of the boot loader - background, fonts, and colors of specific elements.

Additional resources

- For additional information about customizing the boot menu, see *GNU GRUB Manual 2.00*.
- For more general information about **GRUB2**, see *Managing, monitoring and updating the kernel*.

CHAPTER 4. BRANDING AND CHROMING THE GRAPHICAL USER INTERFACE

The customization of Anaconda user interface may include the customization of graphical elements and the customization of product name.

This section provides information about how to customize the graphical elements and the product name.

Prerequisites

1. You have downloaded and extracted the ISO image.
2. You have created your own branding material.

For information about downloading and extracting boot images, see Extracting Red Hat Enterprise Linux boot images

The user interface customization involves the following high-level tasks:

1. Complete the prerequisites.
2. Create custom branding material (if you plan to customize the graphical elements)
3. Customize the graphical elements (if you plan to customize it)
4. Customize the product name (if you plan to customize it)
5. Create a product.img file
6. Create a custom Boot image

NOTE

To create the custom branding material, first refer to the default graphical element files type and dimensions. You can accordingly create the custom material. Details about default graphical elements are available in the sample files that are provided in the Customizing graphical elements section.

4.1. CUSTOMIZING GRAPHICAL ELEMENTS

To customize the graphical elements, you can modify or replace the customisable elements with the custom branded material, and update the container files.

The customisable graphical elements of the installer are stored in the **/usr/share/anaconda/pixmaps/** directory in the installer runtime file system. This directory contains the following customisable files:

```
pixmaps
├── anaconda-password-show-off.svg
├── anaconda-password-show-on.svg
├── right-arrow-icon.png
├── sidebar-bg.png
├── sidebar-logo.png
└── topbar-bg.png
```

Additionally, the **/usr/share/anaconda/** directory contains a CSS stylesheet named **anaconda-gtk.css**, which determines the file names and parameters of the main UI elements - the logo and the backgrounds for the sidebar and top bar. The file has the following contents that can be customized as per your requirement:

```
/* theme colors/images */

@define-color product_bg_color @redhat;

/* logo and sidebar classes */

.logo-sidebar {
  background-image: url('/usr/share/anaconda/pixmaps/sidebar-bg.png');
  background-color: @product_bg_color;
  background-repeat: no-repeat;
}

/* Add a logo to the sidebar */

.logo {
  background-image: url('/usr/share/anaconda/pixmaps/sidebar-logo.png');
  background-position: 50% 20px;
  background-repeat: no-repeat;
  background-color: transparent;
}

/* This is a placeholder to be filled by a product-specific logo. */

.product-logo {
  background-image: none;
  background-color: transparent;
}

AnacondaSpokeWindow #nav-box {
  background-color: @product_bg_color;
  background-image: url('/usr/share/anaconda/pixmaps/topbar-bg.png');
  background-repeat: no-repeat;
  color: white;
}
```

The most important part of the CSS file is the way in which it handles scaling based on resolution. The PNG image backgrounds do not scale, they are always displayed in their true dimensions. Instead, the backgrounds have a transparent background, and the stylesheet defines a matching background color on the **@define-color** line. Therefore, the background *images* "fade" into the background *color*, which means that the backgrounds work on all resolutions without a need for image scaling.

You could also change the **background-repeat** parameters to tile the background, or, if you are confident that every system you will be installing on will have the same display resolution, you can use background images which fill the entire bar.

Any of the files listed above can be customized. Once you do so, follow the instructions in Section 2.2, “Creating a product.img File” to create your own product.img with custom graphics, and then Section 2.3, “Creating Custom Boot Images” to create a new bootable ISO image with your changes included.

4.2. CUSTOMIZING THE PRODUCT NAME

To customize the product name, you must create a custom **.buildstamp file**. To do so, create a new file **.buildstamp.py** with the following content:

```
[Main]
Product=My Distribution
Version=9
BugURL=https://bugzilla.redhat.com/
IsFinal=True
UUID=202007011344.x86_64
[Compose]
Lorax=28.14.49-1
```

Change *My Distribution* to the name which you want to display in the installer.

After you create the custom .buildstamp file, follow the steps in Creating a product.img file section to create a new product.img file containing your customizations, and the Creating custom boot images section to create a new bootable ISO file with your changes included.

4.3. CUSTOMIZING THE DEFAULT CONFIGURATION

You can create your own configuration file and use it to customize the configuration of the installer.

4.3.1. Configuring the default configuration files

You can write the Anaconda configuration files in the **.ini** file format. The Anaconda configuration file consists of sections, options and comments. Each section is defined by a **[section]** header, the comments starting with a **#** character and the keys to define the **options**. The resulting configuration file is processed with the **configparser** configuration file parser.

The default configuration file, located at **/etc/anaconda/anaconda.conf**, contains the documented sections and options that are supported. The file provides a full default configuration of the installer. You can modify the configuration of the product configuration files from **/etc/anaconda/product.d/** and the custom configuration files from **/etc/anaconda/conf.d/**.

The following configuration file describes the default configuration of RHEL 9:

```
[Anaconda]
# Run Anaconda in the debugging mode.
debug = False

# Enable Anaconda addons.
# This option is deprecated and will be removed in the future.
# addons_enabled = True

# List of enabled Anaconda DBus modules.
# This option is deprecated and will be removed in the future.
# kickstart_modules =

# List of Anaconda DBus modules that can be activated.
# Supported patterns: MODULE.PREFIX., MODULE.NAME activatable_modules =
org.fedoraproject.Anaconda.Modules.
    org.fedoraproject.Anaconda.Addons.*

# List of Anaconda DBus modules that are not allowed to run.
# Supported patterns: MODULE.PREFIX., MODULE.NAME forbidden_modules = # List of
```

```
Anaconda DBus modules that can fail to run. # The installation won't be aborted because of
them. # Supported patterns: MODULE.PREFIX., MODULE.NAME
optional_modules =
    org.fedoraproject.Anaconda.Modules.Subscription
    org.fedoraproject.Anaconda.Addons.*


[Installation System]

# Should the installer show a warning about enabled SMT?
can_detect_enabled_smt = False


[Installation Target]
# Type of the installation target.
type = HARDWARE

# A path to the physical root of the target.
physical_root = /mnt/sysimage

# A path to the system root of the target.
system_root = /mnt/sysroot

# Should we install the network configuration?
can_configure_network = True


[Network]
# Network device to be activated on boot if none was configured so.
# Valid values:
#
#   NONE                  No device
#   DEFAULT_ROUTE_DEVICE   A default route device
#   FIRST_WIRED_WITH_LINK  The first wired device with link
#
default_on_boot = NONE


[Payload]
# Default package environment.
default_environment =

# List of ignored packages.
ignored_packages =

# Names of repositories that provide latest updates.
updates_repositories =

# List of .treeinfo variant types to enable.
# Valid items:
#
#   addon
#   optional
#   variant
#
enabled_repositories_from_treeinfo = addon optional variant
```

```
# Enable installation from the closest mirror.
enable_closest_mirror = True

# Default installation source.
# Valid values:
#
#    CLOSEST_MIRROR  Use closest public repository mirror.
#    CDN             Use Content Delivery Network (CDN).
#
default_source = CLOSEST_MIRROR

# Enable ssl verification for all HTTP connection
verify_ssl = True

# GPG keys to import to RPM database by default.
# Specify paths on the installed system, each on a line.
# Substitutions for $releasever and $basearch happen automatically.
default_rpm_gpg_keys =

[Security]
# Enable SELinux usage in the installed system.
# Valid values:
#
#  -1  The value is not set.
#   0  SELinux is disabled.
#   1  SELinux is enabled.
#
selinux = -1


[Bootloader]
# Type of the bootloader.
# Supported values:
#
#   DEFAULT   Choose the type by platform.
#   EXTLINUX  Use extlinux as the bootloader.
#
type = DEFAULT

# Name of the EFI directory.
efi_dir = default

# Hide the GRUB menu.
menu_auto_hide = False

# Are non-iBFT iSCSI disks allowed?
nonibft_iscsi_boot = False

# Arguments preserved from the installation system.
preserved_arguments =
    cio_ignore rd.znet rd_ZNET zfcp.allow_lun_scan
    speakup_synth apic noapic apm ide noht acpi video
    pci nodmraid nompath nomodeset noiswmd fips selinux
    biosdevname ipv6.disable net.ifnames net.ifnames.prefix
    nosmt
```

```
[Storage]
# Enable dmraid usage during the installation.
dmraid = True

# Enable iBFT usage during the installation.
ibft = True

# Do you prefer creation of GPT disk labels?
gpt = False

# Tell multipathd to use user friendly names when naming devices during the installation.
multipath_friendly_names = True

# Do you want to allow imperfect devices (for example, degraded mdraid array devices)?
allow_imperfect_devices = False

# Default file system type. Use whatever Blivet uses by default.
file_system_type =

# Default partitioning.
# Specify a mount point and its attributes on each line.
#
# Valid attributes:
#
#   size <SIZE>    The size of the mount point.
#   min <MIN_SIZE> The size will grow from MIN_SIZE to MAX_SIZE.
#   max <MAX_SIZE> The max size is unlimited by default.
#   free <SIZE>    The required available space.
#
default_partitioning =
   /     (min 1 GiB, max 70 GiB)
   /home (min 500 MiB, free 50 GiB)

# Default partitioning scheme.
# Valid values:
#
#   PLAIN      Create standard partitions.
#   BTRFS      Use the Btrfs scheme.
#   LVM        Use the LVM scheme.
#   LVM_THINP  Use LVM Thin Provisioning.
#
default_scheme = LVM

# Default version of LUKS.
# Valid values:
#
#   luks1  Use version 1 by default.
#   luks2  Use version 2 by default.
#
luks_version = luks2


[Storage Constraints]
```

```
# Minimal size of the total memory.
min_ram = 320 MiB

# Minimal size of the available memory for LUKS2.
luks2_min_ram = 128 MiB

# Should we recommend to specify a swap partition?
swap_is_recommended = False

# Recommended minimal sizes of partitions.
# Specify a mount point and a size on each line.
min_partition_sizes =
    /      250 MiB
    /usr   250 MiB
    /tmp   50  MiB
    /var   384 MiB
    /home  100 MiB
    /boot  200 MiB

# Required minimal sizes of partitions.
# Specify a mount point and a size on each line.
req_partition_sizes =

# Allowed device types of the / partition if any.
# Valid values:
#
#   LVM         Allow LVM.
#   MD          Allow RAID.
#   PARTITION   Allow standard partitions.
#   BTRFS       Allow Btrfs.
#   DISK        Allow disks.
#   LVM_THINP   Allow LVM Thin Provisioning.
#
root_device_types =

# Mount points that must be on a linux file system.
# Specify a list of mount points.
must_be_on_linuxfs = / /var /tmp /usr /home /usr/share /usr/lib

# Paths that must be directories on the / file system.
# Specify a list of paths.
must_be_on_root = /bin /dev /sbin /etc /lib /root /mnt lost+found /proc

# Paths that must NOT be directories on the / file system.
# Specify a list of paths.
must_not_be_on_root =

# Mount points that are recommended to be reformatted.
#
# It will be recommended to create a new file system on a mount point
# that has an allowed prefix, but does not have a blocked one.
# Specify lists of mount points.
reformat_allowlist = /boot /var /tmp /usr
reformat_blocklist = /home /usr/local /opt /var/www
```

```
[User Interface]
# The path to a custom stylesheet.
custom_stylesheet =

# The path to a directory with help files.
help_directory = /usr/share/anaconda/help

# A list of spokes to hide in UI.
# FIXME: Use other identification then names of the spokes.
hidden_spokes =

# Should the UI allow to change the configured root account?
can_change_root = False

# Should the UI allow to change the configured user accounts?
can_change_users = False

# Define the default password policies.
# Specify a policy name and its attributes on each line.
#
# Valid attributes:
#
#   quality <NUMBER>  The minimum quality score (see libpwquality).
#   length <NUMBER>   The minimum length of the password.
#   empty             Allow an empty password.
#   strict            Require the minimum quality.
#
password_policies =
    root (quality 1, length 6)
    user (quality 1, length 6, empty)
    luks (quality 1, length 6)


[License]
# A path to EULA (if any)
#
# If the given distribution has an EULA & feels the need to
# tell the user about it fill in this variable by a path
# pointing to a file with the EULA on the installed system.
#
# This is currently used just to show the path to the file to
# the user at the end of the installation.
eula =
```

4.3.2. Configuring the product configuration files

The product configuration files have one or two extra sections that identify the product. The **[Product]** section specifies the product name of a product. The **[Base Product]** section specifies the product name of a base product if any. For example, Red Hat Enterprise Linux is a base product of Red Hat Virtualization.

The installer loads configuration files of the base products before it loads the configuration file of the specified product. For example, it will first load the configuration for Red Hat Enterprise Linux and then the configuration for Red Hat Virtualization.

See an example of the product configuration file for Red Hat Enterprise Linux:

```
# Anaconda configuration file for Red Hat Enterprise Linux.

[Product]
product_name = Red Hat Enterprise Linux

[Installation System]

# Show a warning if SMT is enabled.
can_detect_enabled_smt = True

[Network]
default_on_boot = DEFAULT_ROUTE_DEVICE

[Payload]
ignored_packages =
    ntfsprogs
    btrfs-progs
    dmraid

enable_closest_mirror = False
default_source = CDN

[Bootloader]
efi_dir = redhat

[Storage]
file_system_type = xfs
default_partitioning =
    /     (min 1 GiB, max 70 GiB)
    /home (min 500 MiB, free 50 GiB)
    swap

[Storage Constraints]
swap_is_recommended = True

[User Interface]

help_directory = /usr/share/anaconda/help/rhel

[License]
eula = /usr/share/redhat-release/EULA
```

See an example of the product configuration file for Red Hat Virtualization:

```
# Anaconda configuration file for Red Hat Virtualization.

[Product]
product_name = Red Hat Virtualization (RHVH)

[Base Product]
product_name = Red Hat Enterprise Linux

[Storage]
default_scheme = LVM_THINP
default_partitioning =
```

```
    /             (min 6 GiB)
    /home         (size 1 GiB)
    /tmp          (size 1 GiB)
    /var          (size 15 GiB)
    /var/crash    (size 10 GiB)
    /var/log      (size 8 GiB)
    /var/log/audit (size 2 GiB)
    swap

[Storage Constraints]
root_device_types = LVM_THINP
must_not_be_on_root = /var
req_partition_sizes =
 /var   10 GiB
 /boot  1  GiB
```

To customize the installer configuration for your product, you must create a product configuration file. Create a new file named **my-distribution.conf**, with content similar to the example above. Change *product_name* in the **[Product]** section to the name of your product, for example My Distribution. The product name should be the same as the name used in the **.buildstamp** file.

After you create the custom configuration file, follow the steps in Creating a product.img file section to create a **new product.img** file containing your customizations, and the Creating custom boot images to create a new bootable ISO file with your changes included.

4.3.3. Configuring the custom configuration files

To customize the installer configuration independently of the product name, you must create a custom configuration file. To do so, create a new file named **100-my-configuration.conf** with the content similar to the example in Configuring the default configuration files and omit the **[Product]** and **[Base Product]** sections.

After you create the custom configuration file, follow the steps in Creating a product.img file section to create a **new product.img** file containing your customizations, and the Creating custom boot images to create a new bootable ISO file with your changes included.

CHAPTER 5. DEVELOPING INSTALLER ADD-ONS

This section provides details about Anaconda and it's architecture, and how to develop your own add-ons. The details about Anaconda and its architecture helps you to understand Anaconda backend and various plug points for the add-ons to work. It also helps to accordingly develop the add-ons.

5.1. INTRODUCTION TO ANACONDA AND ADD-ONS

Anaconda is the operating system installer used in Fedora, Red Hat Enterprise Linux, and their derivatives. It is a set of Python modules and scripts together with some additional files like **Gtk** widgets (written in C), **systemd** units, and **dracut** libraries. Together, they form a tool that allows users to set parameters of the resulting (target) system and then set up this system on a machine. The installation process has four major steps:

1. Prepare installation destination (usually disk partitioning)
2. Install package and data
3. Install and configure boot loader
4. Configure newly installed system

Using Anaconda enables you to install Fedora, Red Hat Enterprise Linux, and their derivatives, in the following three ways:

Using graphical user interface (GUI):

This is the most common installation method. The interface allows users to install the system interactively with little or no configuration required before starting the installation. This method covers all common use cases, including setting up complicated partitioning layouts.

The graphical interface supports remote access over **VNC**, which allows you to use the GUI even on systems with no graphics cards or attached monitor.

Using text user interface (TUI):

The TUI works similar to a monochrome line printer, which allows it to work on serial consoles that do not support cursor movement, colors and other advanced features. The text mode is limited and allows you to customize only the most common options, such as network settings, language options or installation (package) source; advanced features such as manual partitioning are not available in this interface.

Using Kickstart file:

A Kickstart file is a plain text file with shell-like syntax that can contain data to drive the installation process. A Kickstart file allows you to partially or completely automate the installation. A set of commands which configures all required areas is necessary to completely automate the installation. If one or more commands are missed, the installation requires interaction.

Apart from automation of the installer itself, Kickstart files can contain custom scripts that are run at specific moments during the installation process.

5.2. ANACONDA ARCHITECTURE

Anaconda is a set of Python modules and scripts. It also uses several external packages and libraries. The major components of this toolset include the following packages:

- **pykickstart** - parses and validates the Kickstart files. Also, provides data structure that stores values that drive the installation.
- **dnf** - the package manager that installs packages and resolves dependencies
- **blivet** - handles all activities related to storage management
- **pyanaconda** - contains the user interface and modules for **Anaconda**, such as keyboard and timezone selection, network configuration, and user creation. Also provides various utilities to perform system-oriented functions
- **python-meh** - contains an exception handler that gathers and stores additional system information in case of a crash and passes this information to the **libreport** library, which itself is a part of the ABRT Project
- **dasbus** - enables communication between the **D-Bus** library with modules of anaconda and with external components
- **python-simpleline** - text UI framework library to manage user interaction in the **Anaconda** text mode
- **gtk** - the Gnome toolkit library for creating and managing GUI

Apart from the division into packages previously mentioned, **Anaconda** is internally divided into the user interface and a set of modules that run as separate processes and communicate using the **D-Bus** library. These modules are:

- **Boss** - manages the internal module discovery, lifecycle, and coordination
- **Localization** - manages locales
- **Network** - handles network
- **Payloads** - handles data for installation in different formats, such as **rpm**, **ostree**, **tar** and other installation formats. Payloads manage the sources of data for installation; sources can vary in format such as CD-ROM, HDD, NFS, URLs, and other sources
- **Security** - manages security related aspects
- **Services** - handles services
- **Storage** - manages storage using **blivet**
- **Subscription** - handles the **subscription-manager** tool and Insights.
- **Timezone** - deals with time, date, zones, and time synchronization.
- **Users** - creates users and groups.

Each module declares which parts of Kickstart it handles, and has methods to apply the configuration from Kickstart to the installation environment and to the installed system.

The Python code portion of Anaconda (**pyanaconda**) starts as a "main" process that owns the user interface. Any Kickstart data you provide are parsed using the **pykickstart** module and the **Boss** module is started, it discovers all other modules, and starts them. Main process then sends Kickstart data to the modules according to their declared capabilities. Modules process the data, apply the configuration to

the installation environment, and the UI validates if all required choices have been made. If not, you must supply the data in an interactive installation mode. Once all required choices have been made, the installation can start - the modules write data to the installed system.

5.3. ANACONDA USER INTERFACE

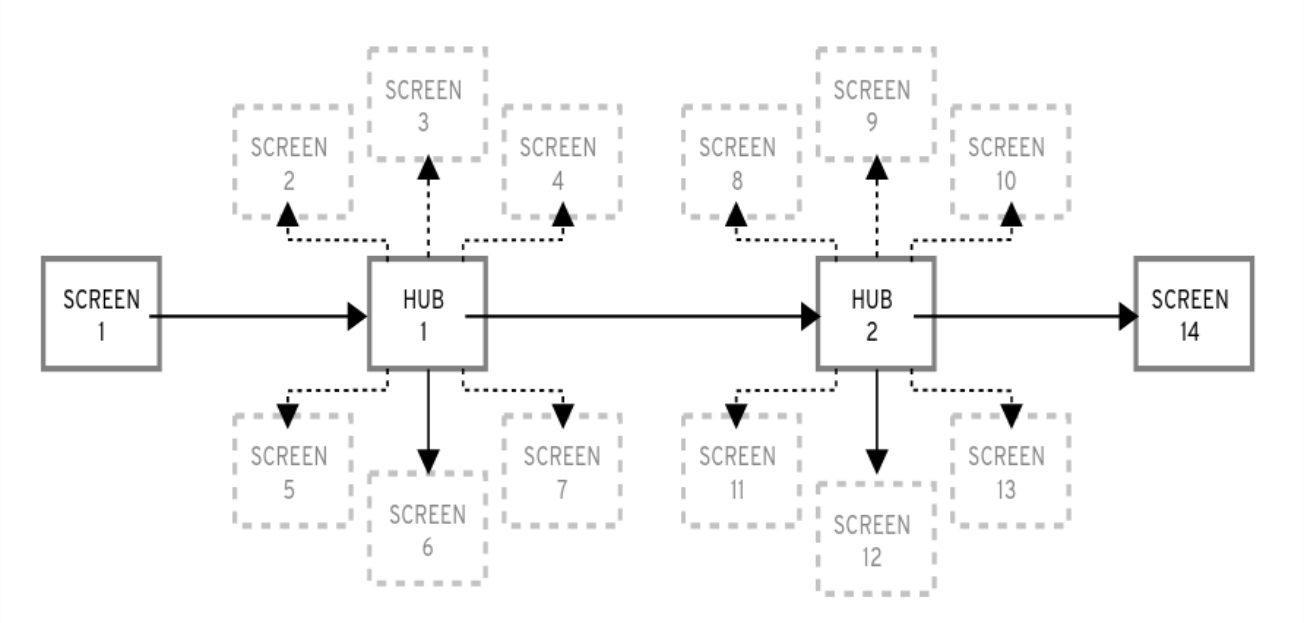
The Anaconda user interface (UI) has a non-linear structure, also known as hub and spoke model.

The advantages of **Anaconda** hub and spoke model are:

- Flexibility to follow the installer screens.
- Flexibility to retain the default settings.
- Provides an overview of the configured values.
- Supports extensibility. You can add hubs without the need to reorder anything and can resolve some complex ordering dependencies.
- Supports installation in graphical and text mode.

The following diagram shows the installer layout and the possible interactions between *hubs* and *spokes* (screens):

Figure 5.1. Hub and spoke model



In the diagram, screens 2-13 are called *normal spokes*, and screens 1 and 14 are *standalone spokes*. Standalone spokes are the screens that can be used before or after the standalone spoke or hub. For example, the **Welcome** screen at the beginning of the installation which prompts you to choose your language for the rest of the installation.

NOTE

- The **Installation Summary** is the only hub in Anaconda. It shows a summary of configured options before the installation begins

Each spoke has the following predefined *properties* that reflect the hub.

- **ready** - states whether or not you can visit a spoke. For example, when the installer is configuring a package source, the spoke is colored in gray, and you cannot access it until the configuration is complete.
- **completed** - marks whether or not the spoke is complete (all required values are set).
- **mandatory** - determines whether you *must* visit the spoke before continuing the installation; for example, you must visit the **Installation Destination** spoke, even if you want to use automatic disk partitioning
- **status** - provides a short summary of values configured within the spoke (displayed under the spoke name in the hub)

To make the user interface clearer, spokes are grouped together into *categories*. For example, the **Localization** category groups together spokes for keyboard layout selection, language support and time zone settings.

Each spoke contains UI controls which display and allow you to modify values from one or more modules. The same applies to spokes that add-ons provide.

5.4. COMMUNICATION ACROSS ANACONDA THREADS

Some of the actions that you need to perform during the installation process may take a long time. For example, scanning disks for existing partitions or downloading package metadata. To prevent you from waiting and remaining responsive, **Anaconda** runs these actions in separate threads.

The **Gtk** toolkit does not support element changes from multiple threads. The main event loop of **Gtk** runs in the main thread of the **Anaconda** process. Therefore, all actions pertaining to the GUI must be performed in the main thread. To do so, use **GLib.idle_add**, which is not always easy or desired. Several helper functions and decorators that are defined in the **pyanaconda.ui.gui.utils** module may add to the difficulty.

The **@gtk_action_wait** and **@gtk_action_nowait** decorators change the decorated function or method in such a way that when this function or method is called, it is automatically queued into Gtk's main loop that runs in the main thread. The return value is either returned to the caller or dropped, respectively.

In a spoke and hub communication, a spoke announces when it is ready and is not blocked. The **hubQ** message queue handles this function, and periodically checks the main event loop. When a spoke becomes accessible, it sends a message to the queue announcing the change and that it should no longer be blocked.

The same applies in a situation where a spoke needs to refresh its status or complete a flag. The **Configuration and Progress** hub has a different queue called **progressQ** which serves as a medium to transfer installation progress updates.

These mechanisms are also used for the text-based interface. In the text mode, there is no main loop, but the keyboard input takes most of the time.

5.5. ANACONDA MODULES AND D-BUS LIBRARY

Anaconda's modules run as independent processes. To communicate with these processes via their **D-Bus** API, use the **dasbus** library.

Calls to methods via **D-Bus** API are asynchronous, but with the **dasbus** library you can convert them to synchronous method calls in Python. You can also write either of the following programs:

- program with asynchronous calls and return handlers
- A program with synchronous calls that makes the caller wait until the call is complete.

For more information about threads and communication, see Communication across Anaconda threads.

Additionally, Anaconda uses Task objects running in modules. Tasks have a **D-Bus** API and methods that are automatically executed in additional threads. To successfully run the tasks, use the **sync_run_task** and **async_run_task** helper functions.

5.6. THE HELLO WORLD ADDON EXAMPLE

Anaconda developers publish an example addon called "Hello World", available on GitHub: https://github.com/rhinstaller/hello-world-anaconda-addon/ The descriptions in further sections are reproduced in this.

5.7. ANACONDA ADD-ON STRUCTURE

An **Anaconda** add-on is a Python package that contains a directory with an **__init__.py** and other source directories (subpackages). Because Python allows you to import each package name only once, specify a unique name for the package top-level directory. You can use an arbitrary name, because add-ons are loaded regardless of their name - the only requirement is that they must be placed in a specific directory.

The suggested naming convention for add-ons is similar to Java packages or D-Bus service names.

To make the directory name a unique identifier for a Python package, prefix the add-on name with the reversed domain name of your organization, using underscores (_) instead of dots. For example, **com_example_hello_world**.

IMPORTANT

Make sure to create an **__init__.py** file in each directory. Directories missing this file are considered as invalid Python packages.

When writing an add-on, ensure the following:

- Support for each interface (graphical interface and text interface) is available in a separate subpackage and these subpackages are named **gui** for the graphical interface and **tui** for the text-based interface.
- The **gui** and **tui** packages contain a **spokes** subpackage. [1]
- Modules contained in the packages have an arbitrary name.
- The **gui**/ and **tui**/ directories contain Python modules with any name.
- There is a service that performs the actual work of the addon. This service can be written in Python or any other language.
- The service implements support for D-Bus and Kickstart.
- The addon contains files that enable automatic startup of the service.

Following is a sample directory structure for an add-on which supports every interface (Kickstart, GUI and TUI):

Example 5.1. Sample add-on structure

```
com_example_hello_world
├── gui
│   ├── init.py
│   └── spokes
│       └── init.py
└── tui
    ├── init.py
    └── spokes
    └── init.py
```

Each package must contain at least one module with an arbitrary name defining the classes that are inherited from one or more classes defined in the API.

NOTE

For all add-ons, follow Python's PEP 8 and PEP 257 guidelines for docstring conventions. There is no consensus on the format of the actual content of docstrings in **Anaconda**; the only requirement is that they are human-readable. If you plan to use auto-generated documentation for your add-on, docstrings should follow the guidelines for the toolkit you use to accomplish this.

You can include a category subpackage if an add-on needs to define a new category, but this is not recommended.

5.8. ANACONDA SERVICES AND CONFIGURATION FILES

Anaconda services and configuration files are included in data/ directory. These files are required to start the add-ons service and to configure D-Bus.

Following are some examples of Anaconda Hello World add-on:

Example 5.2. Example of *addon-name*.conf:

```
<!DOCTYPE busconfig PUBLIC
"-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
      <policy user="root">
            <allow own="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
            <allow send_destination="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
      </policy>
      <policy context="default">
            <deny own="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
            <allow send_destination="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
      </policy>
</busconfig>
```

This file must be placed in the **/usr/share/anaconda/dbus/confs/** directory in the installation environment. The string **org.fedoraproject.Anaconda.Addons.HelloWorld** must correspond to the location of addon's service on D-Bus.

Example 5.3. Example of *addon-name*.service:

```
[D-BUS Service]
# Start the org.fedoraproject.Anaconda.Addons.HelloWorld service.
# Runs org_fedora_hello_world/service/main.py
Name=org.fedoraproject.Anaconda.Addons.HelloWorld
Exec=/usr/libexec/anaconda/start-module org_fedora_hello_world.service
User=root
```

This file must be placed in the **/usr/share/anaconda/dbus/services/** directory in the installation environment. The string **org.fedoraproject.Anaconda.Addons.HelloWorld** must correspond to the location of addon's service on D-Bus. The value on the line starting with **Exec=** must be a valid command that starts the service in the installation environment.

5.9. GUI ADD-ON BASIC FEATURES

Similarly to Kickstart support in add-ons, GUI support requires that every part of the add-on must contain at least one module with a definition of a class inherited from a particular class defined by the API. For the graphical add-on support, the only class you should add is the **NormalSpoke** class, defined in **pyanaconda.ui.gui.spokes**, as a class for the normal spoke type of screen. To learn more about it, see Anaconda user interface.

To implement a new class inherited from **NormalSpoke**, you must define the following class attributes that the API requires:

- **builderObjects** - lists all top-level objects from the spoke's **.glade** file that should be exposed to the spoke with their children objects (recursively). In case everything should be exposed to the spoke, which is not recommended, the list should be empty.
- **mainWidgetName** - contains the id of the main window widget (Add Link) as defined in the **.glade** file.
- **uiFile** - contains the name of the **.glade** file.
- **category** - contains the class of the category the spoke belongs to.
- **icon** - contains the identifier of the icon that will be used for the spoke on the hub.
- **title** - defines the title that will be used for the spoke on the hub.

5.10. ADDING SUPPORT FOR THE ADD-ON GRAPHICAL USER INTERFACE (GUI)

This section describes how to add support to the graphical user interface (GUI) of your add-on by performing the following high-level steps:

1. Define Attributes Required for the Normalspoke Class
2. Define the **__init__** and **initialize** Methods

3. Define the **refresh**, **apply**, and **execute** Methods
4. Define the **status** and the **ready**, **completed** and **mandatory** Properties

Prerequisites

- Your add-on includes support for Kickstart. See Anaconda add-on structure.
- Install the anaconda-widgets and anaconda-widgets-devel packages, which contain Gtk widgets specific for **Anaconda**, such as **SpokeWindow**.

Procedure

- Create the following modules with all required definitions to add support for the Add-on graphical user interface (GUI), according to the following examples.

Example 5.4. Defining Attributes Required for the Normalspoke Class:

```
# will never be translated
_ = lambda x: x
N_ = lambda x: x

# the path to addons is in sys.path so we can import things from org_fedora_hello_world
from org_fedora_hello_world.gui.categories.hello_world import HelloWorldCategory
from pyanaconda.ui.gui.spokes import NormalSpoke

# export only the spoke, no helper functions, classes or constants
__all__ = ["HelloWorldSpoke"]

class HelloWorldSpoke(FirstbootSpokeMixIn, NormalSpoke):
    """
    Class for the Hello world spoke. This spoke will be in the Hello world
    category and thus on the Summary hub. It is a very simple example of a unit
    for the Anaconda's graphical user interface. Since it is also inherited form
    the FirstbootSpokeMixIn, it will also appear in the Initial Setup (successor
    of the Firstboot tool).

    :see: pyanaconda.ui.common.UIObject
    :see: pyanaconda.ui.common.Spoke
    :see: pyanaconda.ui.gui.GUIObject
    :see: pyanaconda.ui.common.FirstbootSpokeMixIn
    :see: pyanaconda.ui.gui.spokes.NormalSpoke

    """

    # class attributes defined by API #

    # list all top-level objects from the .glade file that should be exposed
    # to the spoke or leave empty to extract everything
    builderObjects = ["helloWorldSpokeWindow", "buttonImage"]

    # the name of the main window widget
    mainWidgetName = "helloWorldSpokeWindow"

    # name of the .glade file in the same directory as this source
```

```
    uiFile = "hello_world.glade"

    # category this spoke belongs to
    category = HelloWorldCategory

    # spoke icon (will be displayed on the hub)
    # preferred are the -symbolic icons as these are used in Anaconda's spokes
    icon = "face-cool-symbolic"

    # title of the spoke (will be displayed on the hub)
    title = N_("_HELLO WORLD")
```

The **__all__** attribute exports the **spoke** class, followed by the first lines of its definition including definitions of attributes previously mentioned in GUI Add-on basic features. These attribute values are referencing widgets defined in the **com_example_hello_world/gui/spokes/hello.glade** file. Two other notable attributes are present:

- **category**, which has its value imported from the **HelloWorldCategory** class from the **com_example_hello_world.gui.categories** module. The **HelloWorldCategory** that the path to add-ons is in **sys.path** so that values can be imported from the **com_example_hello_world** package. The **category** attribute is part of the **N_ function** name, which marks the string for translation; but returns the non-translated version of the string, as the translation happens in a later stage.

- **title**, which contains one underscore in its definition. The **title** attribute underscore marks the beginning of the title itself and makes the spoke reachable by using the **Alt+H** keyboard shortcut.

What usually follows the header of the class definition and the class **attributes** definitions is the constructor that initializes an instance of the class. In case of the Anaconda graphical interface objects, there are two methods initializing a new instance: the **__init__** method and the **initialize** method.

The reason behind two such functions is that the GUI objects may be created in memory at one time and fully initialized at a different time, as the **spoke** initialization could be time consuming. Therefore, the **__init__** method should only call the parent's **__init__** method and, for example, initialize non-GUI attributes. On the other hand, the **initialize** method that is called when the installer's graphical user interface initializes should finish the full initialization of the spoke.

In the **Hello World add-on** example, define these two methods as follows. Note the number and description of the arguments passed to the **__init__** method.

Example 5.5. Defining the __init__ and initialize Methods:

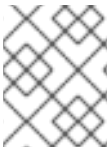
```
def __init__(self, data, storage, payload):
    """
    :see: pyanaconda.ui.common.Spoke.init
    :param data: data object passed to every spoke to load/store data
    from/to it
    :type data: pykickstart.base.BaseHandler
    :param storage: object storing storage-related information
    (disks, partitioning, bootloader, etc.)
    :type storage: blivet.Blivet
    :param payload: object storing packaging-related information
    :type payload: pyanaconda.packaging.Payload
```

```
        """

        NormalSpoke.init(self, data, storage, payload)
        self._hello_world_module = HELLO_WORLD.get_proxy()

    def initialize(self):
        """
        The initialize method that is called after the instance is created.
        The difference between init and this method is that this may take
        a long time and thus could be called in a separate thread.
        :see: pyanaconda.ui.common.UIObject.initialize
        """
        NormalSpoke.initialize(self)
        self._entry = self.builder.get_object("textLines")
        self._reverse = self.builder.get_object("reverseCheckButton")
```

The data parameter passed to the **__init__** method is the in-memory tree-like representation of the Kickstart file where all data is stored. In one of the ancestors' **__init__** methods it is stored in the **self.data** attribute, which allows all other methods in the class to read and modify the structure.



NOTE

The **storage object** is no longer usable as of RHEL9. If your add-on needs to interact with storage configuration, use the **Storage DBus** module.

Because the HelloWorldData class has already been defined in The Hello World addon example , there already is a subtree in self.data for this add-on. Its root, an instance of the class, is available as **self.data.addons.com_example_hello_world**.

Another action that an ancestor's **__init__** does is initializing an instance of the GtkBuilder with the **spoke's .glade** file and storing it as **self.builder**. The **initialize** method uses this to get the **GtkTextEntry** used to show and modify the text from the kickstart file's %addon section.

The **__init__** and **initialize** methods are both important when the spoke is created. However, the main role of the spoke is to be visited by a user who wants to change or review the spoke's values shows and sets. To enable this, three other methods are available:

- **refresh** - called when the spoke is about to be visited; this method refreshes the state of the spoke, mainly its UI elements, to ensure that the displayed data matches internal data structures and, with that, to ensure that current values stored in the self.data structure are displayed.
- **apply** - called when the spoke is left and used to store values from UI elements back into the **self.data** structure.
- **execute** - called when users leave the spoke and used to perform any runtime changes based on the new state of the spoke.

These functions are implemented in the sample Hello World add-on in the following way:

Example 5.6. Defining the refresh, apply and execute Methods

```
def refresh(self):
    """
    The refresh method that is called every time the spoke is displayed.
```

```
        It should update the UI elements according to the contents of
        internal data structures.
        :see: pyanaconda.ui.common.UIObject.refresh
        """
        lines = self._hello_world_module.Lines
        self._entry.get_buffer().set_text("".join(lines))
        reverse = self._hello_world_module.Reverse
        self._reverse.set_active(reverse)

    def apply(self):
        """
        The apply method that is called when user leaves the spoke. It should
        update the D-Bus service with values set in the GUI elements.
        """
        buf = self._entry.get_buffer()
        text = buf.get_text(buf.get_start_iter(),
                            buf.get_end_iter(),
                            True)
        lines = text.splitlines(True)
        self._hello_world_module.SetLines(lines)

        self._hello_world_module.SetReverse(self._reverse.get_active())

    def execute(self):
        """
        The execute method that is called when the spoke is exited. It is
        supposed to do all changes to the runtime environment according to
        the values set in the GUI elements.

        """

        # nothing to do here
        pass
```

You can use several additional methods to control the spoke's state:

- **ready** - determines whether the spoke is ready to be visited; if the value is "False", the **spoke** is not accessible, for example, the **Package Selection** spoke before a package source is configured.
- **completed** - determines if the spoke has been completed.
- **mandatory** - determines if the spoke is mandatory or not, for example, the **Installation Destination** spoke, which must always be visited, even if you want to use automatic partitioning.

All of these attributes need to be dynamically determined based on the current state of the installation process.

Below is a sample implementation of these methods in the Hello World add-on, which requires a certain value to be set in the text attribute of the **HelloWorldData** class:

Example 5.7. Defining the ready, completed and mandatory Methods

```
@property
def ready(self):
```

```
        """
        The ready property reports whether the spoke is ready, that is, can be visited
        or not. The spoke is made (in)sensitive based on the returned value of the ready
        property.

        :rtype: bool

        """

        # this spoke is always ready
        return True


    @property
    def mandatory(self):
        """
        The mandatory property that tells whether the spoke is mandatory to be
        completed to continue in the installation process.

        :rtype: bool

        """

        # this is an optional spoke that is not mandatory to be completed
        return False
```

After these properties are defined, the spoke can control its accessibility and completeness, but it cannot provide a summary of the values configured within - you must visit the spoke to see how it is configured, which may not be desired. For this reason, an additional property called **status** exists. This property contains a single line of text with a short summary of configured values, which can then be displayed in the hub under the spoke title.

The status property is defined in the **Hello World** example add-on as follows:

Example 5.8. Defining the status Property

```
    @property
    def status(self):
        """
        The status property that is a brief string describing the state of the
        spoke. It should describe whether all values are set and if possible
        also the values themselves. The returned value will appear on the hub
        below the spoke's title.
        :rtype: str
        """
        lines = self._hello_world_module.Lines
        if not lines:
            return _("No text added")
        elif self._hello_world_module.Reverse:
            return _("Text set with {} lines to reverse").format(len(lines))
        else:
            return _("Text set with {} lines").format(len(lines))
```

After defining all properties described in the examples, the add-on has full support for showing a graphical user interface (GUI) as well as Kickstart.

NOTE

The example demonstrated here is very simple and does not contain any controls; knowledge of Python Gtk programming is required to develop a functional, interactive spoke in the GUI.

One notable restriction is that each spoke must have its own main window - an instance of the **SpokeWindow** widget. This widget, along with other widgets specific to Anaconda, is found in the **anaconda-widgets** package. You can find other files required for development of add-ons with GUI support, such as **Glade** definitions, in the **anaconda-widgets-devel** package.

Once your graphical interface support module contains all necessary methods you can continue with the following section to add support for the text-based user interface, or you can continue with Deploying and testing an Anaconda add-on and test the add-on.

5.11. ADD-ON GUI ADVANCED FEATURES

The **pyanaconda** package contains several helper and utility functions, as well as constructs which may be used by hubs and spokes. Most of them are located in the **pyanaconda.ui.gui.utils** package.

The sample **Hello World** add-on demonstrates usage of the **englightbox** content manager which **Anaconda** also uses. This content manager can put a window into a lightbox to increase its visibility and focus it to prevent users interacting with the underlying window. To demonstrate this function, the sample add-on contains a button which opens a new dialog window; the dialog itself is a special HelloWorldDialog inheriting from the GUIObject class, which is defined in pyanaconda.ui.gui.*init*.

The dialog class defines the run method that runs and destroys an internal Gtk dialog accessible through the self.window attribute, which is populated using a mainWidgetName class attribute with the same meaning. Therefore, the code defining the dialog is very simple, as demonstrated in the following example:

Example 5.9. Defining a englightbox Dialog

```
# every GUIObject gets ksdata in init
dialog = HelloWorldDialog(self.data)

# show dialog above the lightbox
with self.main_window.enlightbox(dialog.window):
    dialog.run()
```

The **Defining an englightbox Dialog** example code creates an instance of the dialog and then uses the enlightbox context manager to run the dialog within a lightbox. The context manager has a reference to the window of the spoke and only needs the dialog's window to instantiate the lightbox for the dialog.

Another useful feature provided by **Anaconda** is the ability to define a spoke that will appear both during the installation and after the first reboot. The **Initial Setup** utility is described in Adding support for the Add-on graphical user interface (GUI). To make a spoke available in both Anaconda and Initial Setup, it must inherit the special **FirstbootSpokeMixIn** class, also known as **mixin**, as the first inherited class defined in the **pyanaconda.ui.common** module.

To make a spoke available in **Anaconda** and the reconfiguration mode of the Initial Setup, it must inherit the special **FirstbootSpokeMixIn** class, also known as **mixin**, as the first inherited class defined in the **pyanaconda.ui.common** module.

If you want to make a certain spoke available only in Initial Setup, this spoke should instead inherit the **FirstbootOnlySpokeMixIn** class.

To make a spoke always available in both **Anaconda** and Initial Setup, the spoke should redefine the **should_run** method, as demonstrated in the following example:

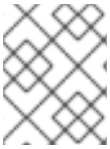
Example 5.10. Redefining the should_run method

```
@classmethod
    def should_run(cls, environment, data):
    """Run this spoke for Anaconda and Initial Setup"""
    return True
```

The **pyanaconda** package provides many more advanced features, such as the **@gtk_action_wait** and **@gtk_action_nowait** decorators, but they are out of scope of this guide. For more examples, refer to the installer's sources.

5.12. TUI ADD-ON BASIC FEATURES

Anaconda also supports a text-based interface (TUI). This interface is more limited in its capabilities, but on some systems it might be the only choice for an interactive installation. For more information about differences between the text-based interface and graphical interface and about limitations of the TUI, see Introduction to Anaconda and add-ons.



NOTE

To add support for the text interface into your add-on, create a new set of subpackages under the tui directory as described in Anaconda add-on structure.

The text mode support in the installer is based on the **simpleline** library, which only allows very simple user interaction. The text mode interface:

- Does not support cursor movement - instead, it acts like a line printer.
- Does not support any visual enhancements, such as using different colors or fonts, for example.

Internally, the **simpleline** toolkit has three main classes: **App**, **UIScreen** and **Widget**. Widgets are units containing information to be printed on the screen. They are placed on UIScreens that are switched by a single instance of the App class. On top of the basic elements, **hubs**, **spoke`s and `dialogs** all contain various widgets in a way similar to the graphical interface.

The most important classes for an add-on are **NormalTUISpoke** and various other classes defined in the **pyanaconda.ui.tui.spokes** package. All those classes are based on the **TUIObject** class, which itself is an equivalent of the **GUIObject** class discussed in Add-on GUI advanced features. Each TUI spoke is a Python class inheriting from the **NormalTUISpoke** class, overriding special arguments and methods defined by the API. Because the text interface is simpler than the GUI, there are only two such arguments:

- **title** - determines the title of the spoke, similar as the title argument in the GUI.

- **category** - determines the category of the spoke as a string; the category name is not displayed anywhere, it is only used for grouping.

NOTE

The TUI handles categories differently than the GUI. It is recommended to assign a pre-existing category to your new spoke. Creating a new category would require patching Anaconda, and brings little benefit.

Each spoke is also expected to override several methods, namely ***init***, **initialize**, **refresh**, **refresh**, **apply**, **execute**, **input**, **prompt**, and **properties** (**ready**, **completed**, **mandatory**, and **status**).

Additional resources

- See Adding support for the Add-on GUI.

5.13. DEFINING A SIMPLE TUI SPOKE

The following example shows the implementation of a simple Text User Interface (TUI) spoke in the Hello World sample add-on:

Prerequisites

- You have created a new set of subpackages under the tui directory as described in Anaconda add-on structure.

Procedure

- Create modules with all required definitions to add support for the add-on text user interface (TUI), according to the following examples:

Example 5.11. Defining a Simple TUI Spoke

```
def __init__(self, *args, **kwargs):
    """
    Create the representation of the spoke.

    :see: simpleline.render.screen.UIScreen
    """
    super().__init__(*args, **kwargs)
    self.title = N_("Hello World")
    self._hello_world_module = HELLO_WORLD.get_proxy()
    self._container = None
    self._reverse = False
    self._lines = ""

def initialize(self):
    """
    The initialize method that is called after the instance is created.
    The difference between __init__ and this method is that this may take
    a long time and thus could be called in a separated thread.

    :see: pyanaconda.ui.common.UIObject.initialize
    """
```

```
        # nothing to do here
        super().initialize()

    def setup(self, args=None):
        """
        The setup method that is called right before the spoke is entered.
        It should update its state according to the contents of DBus modules.

        :see: simpleline.render.screen.UIScreen.setup
        """
        super().setup(args)

        self._reverse = self._hello_world_module.Reverse
        self._lines = self._hello_world_module.Lines

        return True

    def refresh(self, args=None):
        """
        The refresh method that is called every time the spoke is displayed.
        It should generate the UI elements according to its state.

        :see: pyanaconda.ui.common.UIObject.refresh
        :see: simpleline.render.screen.UIScreen.refresh
        """
        super().refresh(args)

        self._container = ListColumnContainer(
            columns=1
        )
        self._container.add(
            CheckboxWidget(
                title="Reverse",
                completed=self._reverse
            ),
            callback=self._change_reverse
        )
        self._container.add(
            EntryWidget(
                title="Hello world text",
                value="".join(self._lines)
            ),
            callback=self._change_lines
        )

        self.window.add_with_separator(self._container)

    def _change_reverse(self, data):
        """
        Callback when user wants to switch checkbox.
        Flip state of the "reverse" parameter which is boolean.
        """
        self._reverse = not self._reverse

    def _change_lines(self, data):
        """
```

```
        Callback when user wants to input new lines.
        Show a dialog and save the provided lines.
        """
        dialog = Dialog("Lines")
        result = dialog.run()
        self._lines = result.splitlines(True)

    def input(self, args, key):
        """
        The input method that is called by the main loop on user's input.

        * If the input should not be handled here, return it.
        * If the input is invalid, return InputState.DISCARDED.
        * If the input is handled and the current screen should be refreshed,
          return InputState.PROCESSED_AND_REDRAW.
        * If the input is handled and the current screen should be closed,
          return InputState.PROCESSED_AND_CLOSE.

        :see: simpleline.render.screen.UIScreen.input
        """
        if self._container.process_user_input(key):
            return InputState.PROCESSED_AND_REDRAW

        if key.lower() == Prompt.CONTINUE:
            self.apply()
            self.execute()
            return InputState.PROCESSED_AND_CLOSE

        return super().input(args, key)

    def apply(self):
        """
        The apply method is not called automatically for TUI. It should be called
        in input() if required. It should update the contents of internal data
        structures with values set in the spoke.
        """
        self._hello_world_module.SetReverse(self._reverse)
        self._hello_world_module.SetLines(self._lines)

    def execute(self):
        """
        The execute method is not called automatically for TUI. It should be called
        in input() if required. It is supposed to do all changes to the runtime
        environment according to the values set in the spoke.
        """
        # nothing to do here
        pass
```

NOTE

It is not necessary to override the ***init*** method if it only calls the ancestor's ***init***, but the comments in the example describe the arguments passed to constructors of spoke classes in an understandable way.

In the previous example:

- The **setup** method sets up a default value for the internal attribute of the spoke on every entry, which is then displayed by the **refresh** method, updated by the **input** method and used by the **apply** method to update internal data structures.
- The **execute** method has the same purpose as the equivalent method in the GUI; in this case, the method has no effect.
- The **input** method is specific to the text interface; there are no equivalents in Kickstart or GUI. The **input** methods are responsible for user interaction.
- The **input** method processes the entered string and takes action depending on its type and value. The above example asks for any value and then stores it as an internal attribute (key). In more complex add-ons, you typically need to perform some non-trivial actions, such as parse letters as actions, convert numbers into integers, show additional screens or toggle boolean values.
- The **return** value of the input class must be either the **InputState** enum or the **input** string itself, in case this input should be processed by a different screen. In contrast to the graphical mode, the **apply** and **execute** methods are not called automatically when leaving the spoke; they must be called explicitly from the input method. The same applies to closing (hiding) the spoke's screen: it must be called explicitly from the **close** method.

To show another screen, for example if you need additional information that was entered in a different spoke, you can instantiate another **TUIObject** and use **ScreenHandler.push_screen_modal()** to show it.

Due to restrictions of the text-based interface, TUI spokes tend to have a very similar structure, that consists of a list of checkboxes or entries that should be checked or unchecked and populated by the user.

5.14. USING NORMALTUISPOKE TO DEFINE A TEXT INTERFACE SPOKE

The Defining a Simple TUI Spoke example showed a way to implement a TUI spoke where its methods handle printing and processing the available and provided data. However, there is a different way to accomplish this using the **Normal EditTUISpoke** class from the **pyanaconda.ui.tui.spokes** package. By inheriting this class, you can implement a typical TUI spoke by only specifying fields and attributes that should be set in it. The following example demonstrates this:

Prerequisites

- You have added a new set of subpackages under the **TUI** directory, as described in Anaconda add-on structure.

Procedure

- Create modules with all required definitions to add support for the Add-on text user interface (TUI), according to the following examples.

Example 5.12. Using NormalTUISpoke to Define a Text Interface Spoke

```
class HelloWorldEditSpoke(NormalTUISpoke):
    """Example class demonstrating usage of editing in TUI"""
```

```
category = HelloWorldCategory

def init(self, data, storage, payload):
    """
    :see: simpleline.render.screen.UIScreen
    :param data: data object passed to every spoke to load/store data
                 from/to it
    :type data: pykickstart.base.BaseHandler
    :param storage: object storing storage-related information
                    (disks, partitioning, bootloader, etc.)
    :type storage: blivet.Blivet
    :param payload: object storing packaging-related information
    :type payload: pyanaconda.packaging.Payload
    """
    NormalTUISpoke.init(self, data, storage, payload)

    self.title = N_("Hello World Edit")
    self._container = None
    # values for user to set
    self._checked = False
    self._unconditional_input = ""
    self._conditional_input = ""

def refresh(self, args=None):
    """
    The refresh method that is called every time the spoke is displayed.
    It should update the UI elements according to the contents of
    self.data.
    :see: pyanaconda.ui.common.UIObject.refresh
    :see: simpleline.render.screen.UIScreen.refresh
    :param args: optional argument that may be used when the screen is
                 scheduled
    :type args: anything
    """
    super().refresh(args)
    self._container = ListColumnContainer(columns=1)

    # add ListColumnContainer to window (main window container)
    # this will automatically add numbering and will call callbacks when required
    self.window.add(self._container)

    self._container.add(CheckboxWidget(title="Simple checkbox", completed=self._checked),
                        callback=self._checkbox_called)
    self._container.add(EntryWidget(title="Unconditional text input",
                                    value=self._unconditional_input),
                        callback=self._get_unconditional_input)

    # show conditional input only if the checkbox is checked
    if self._checked:
        self._container.add(EntryWidget(title="Conditional password input",
                                        value="Password set" if self._conditional_input
                                        else ""),
                            callback=self._get_conditional_input)

    self._window.add_separator()
```

```
@property
def completed(self):
    # completed if user entered something non-empty to the Conditioned input
    return bool(self._conditional_input)
@property
def status(self):
    return "Hidden input %s" % ("entered" if self._conditional_input
                                else "not entered")

def apply(self):
    # nothing needed here, values are set in the self.args tree
    pass
```

5.15. DEPLOYING AND TESTING AN ANACONDA ADD-ON

You can deploy and test your own Anaconda add-on into the installation environment. To do so, follow the steps:

Prerequisites

- You created an Add-on.
- You have access to your **D-Bus** files.

Procedure

1. Create a directory ***DIR*** at the place of your preference.
2. Add the **Add-on** python files into ***DIR*****/usr/share/anaconda/addons/**.
3. Copy your **D-Bus** service file into ***DIR*****/usr/share/anaconda/dbus/services/**.
4. Copy your **D-Bus** service configuration file to **/usr/share/anaconda/dbus/confs/**.
5. Create the *updates* image.
   Access the ***DIR*** directory:

   ```
   cd DIR
   ```

   Locate the *updates* image.

   ```
   find . | cpio -c -o | pigz -9cv > DIR/updates.img
   ```

6. Extract the contents of the ISO boot image.
7. Use the resulting ***updates*** image:
   a. Add the ***updates*.img** file into the **images** directory of your unpacked ISO contents.
   b. Repack the image.

c. Set up a web server to provide the ***updates***.img file to the Anaconda installer via HTTP.

d. Load ***updates***.img file at boot time by adding the following specification to the boot options.

```
inst.updates=http://your-server/whatever/updates.img to boot options.
```

For specific instructions on unpacking an existing boot image, creating a **product.img** file and repackaging the image, see Extracting Red Hat Enterprise Linux boot images .

---

[1] The **gui** package may also contain a **categories** subpackage if the add-on needs to define a new category, but this is not recommended.

CHAPTER 6. COMPLETING POST CUSTOMIZATION TASKS

To complete the customizations made, perform the following tasks:

- Create a product.img image file (applies only for graphical customizations).
- Create a custom boot image.

This section provides information about how to create a product.img image file and to create a custom boot image.

6.1. CREATING A PRODUCT.IMG FILE

A **product.img** image file is an archive containing new installer files that replace the existing ones at runtime.

During a system boot, **Anaconda** loads the product.img file from the images/ directory on the boot media. It then uses the files that are present in this directory to replace identically named files in the installer's file system. The files when replaced customizes the installer (for example, for replacing default images with custom ones).

Note: The **product.img** image must contain a directory structure identical to the installer. For more information about the installer directory structure, see the table below.

Table 6.1. Installer directory structure and custom contents

Type of custom content	File system location
Pixmaps (logo, sidebar, top bar, and so on.)	**/usr/share/anaconda/pixmaps/**
GUI stylesheet	**/usr/share/anaconda/anaconda-gtk.css**
Anaconda add-ons	**/usr/share/anaconda/addons/**
Product configuration files	**/etc/anaconda/product.d/**
Custom configuration files	**/etc/anaconda/conf.d/**
Anaconda DBus service conf files	**/usr/share/anaconda/dbus/confs/**
Anaconda DBus service files	**/usr/share/anaconda/dbus/services/**

The procedure below explains how to create a **product.img** file.

Procedure

1. Navigate to a working directory such as **/tmp**, and create a subdirectory named **product/**:

```
$ cd /tmp
```

2. Create a subdirectory product/

```
$ mkdir product/
```

3. Create a directory structure identical to the location of the file you want to replace. For example, if you want to test an add-on that is present in the **/usr/share/anaconda/addons** directory on the installation system, create the same structure in your working directory:

```
$ mkdir -p product/usr/share/anaconda/addons
```

NOTE

To view the installer's runtime file, boot the installation and switch to virtual console 1 (**Ctrl**+**Alt**+**F1**) and then switch to the second **tmux** window (**Ctrl**+**b**+**2**). A shell prompt that can be used to browse a file system opens.

4. Place your customized files (in this example, custom add-on for **Anaconda**) into the newly created directory:

```
$ cp -r ~/path/to/custom/addon/ product/usr/share/anaconda/addons/
```

5. Repeat steps 3 and 4 (create a directory structure and place the custom files into it) for every file you want to add to the installer.

6. Create a **.buildstamp** file in the root of the directory. The **.buildstamp** file describes the system version, the product and several other parameters. The following is an example of a **.buildstamp** file from Red Hat Enterprise Linux 8.4:

```
[Main]
Product=Red Hat Enterprise Linux
Version=8.4
BugURL=https://bugzilla.redhat.com/
IsFinal=True
UUID=202007011344.x86_64
[Compose]
Lorax=28.14.49-1
```

The **IsFinal** parameter specifies whether the image is for a release (GA) version of the product (**True**), or a pre-release such as Alpha, Beta, or an internal milestone (**False**).

7. Navigate to the **product/** directory, and create the **product.img** archive:

```
$ cd product
```

```
$ find . | cpio -c -o | gzip -9cv > ../product.img
```

This creates a **product.img** file one level above the **product/** directory.

8. Move the **product.img** file to the **images/** directory of the extracted ISO image.

The product.img file is now created and the customizations that you want to make are placed in the respective directories.

NOTE

Instead of adding the **product.img** file on the boot media, you can place this file into a different location and use the **inst.updates=** boot option at the boot menu to load it. In that case, the image file can have any name, and it can be placed in any location (USB flash drive, hard disk, HTTP, FTP or NFS server), as long as this location is reachable from the installation system.

6.2. CREATING CUSTOM BOOT IMAGES

After you customize the boot images and the GUI layout, create a new image that includes the changes you made.

To create custom boot images, follow the procedure below.

Procedure

1. Make sure that all of your changes are included in the working directory. For example, if you are testing an add-on, make sure to place the **product.img** in the **images/** directory.

2. Make sure your current working directory is the top-level directory of the extracted ISO image, for example, **/tmp/ISO/iso/**.

3. Create a new ISO image using the **genisoimage**:

```
# genisoimage -U -r -v -T -J -joliet-long -V "RHEL-9 Server.x86_64" -volset "RHEL-9 Server.x86_64" -A "RHEL-9 Server.x86_64" -b isolinux/isolinux.bin -c isolinux/boot.cat -no-emul-boot -boot-load-size 4 -boot-info-table -eltorito-alt-boot -e images/efiboot.img -no-emul-boot -o ../NEWISO.iso .
```

   In the above example:

   - Make sure that the values for **-V**, **-volset**, and **-A** options match the image's boot loader configuration, if you are using the **LABEL=** directive for options that require a location to load a file on the same disk. If your boot loader configuration (**isolinux/isolinux.cfg** for BIOS and **EFI/BOOT/grub.cfg** for UEFI) uses the **inst.stage2=LABEL=*disk_label*** stanza to load the second stage of the installer from the same disk, then the disk labels must match.

     IMPORTANT

     In boot loader configuration files, replace all spaces in disk labels with **\x20**. For example, if you create an ISO image with a **RHEL 9.0** label, boot loader configuration should use **RHEL\x209.0**.

   - Replace the value of the **-o** option (**-o ../NEWISO.iso**) with the file name of your new image. The value in the example creates the **NEWISO.iso** file in the directory *above* the current one.
     For more information about this command, see the **genisoimage(1)** man page.

4. Implant an MD5 checksum into the image. Note that without an MD5 checksu, the image verification check might fail (the **rd.live.check** option in the boot loader configuration) and the installation can hang.

```
# implantisomd5 ../NEWISO.iso
```

In the above example, replace *../NEWISO.iso* with the file name and the location of the ISO image that you have created in the previous step.

You can now write the new ISO image to physical media or a network server to boot it on physical hardware, or you can use it to start installing a virtual machine.

Additional resources

- For instructions on preparing boot media or network server, see Performing an advanced RHEL 9 installation
- For instructions on creating virtual machines with ISO images, see Configuring and Managing Virtualization.