Red Hat Enterprise Linux 7 Anaconda Customization Guide

Customizing and enhancing the installer

Vratislav Podzimek Petr Bokoč

Red Hat Enterprise Linux 7 Anaconda Customization Guide

Customizing and enhancing the installer

Vratislav Podzimek
vpodzime@redhat.com

Petr Bokoč
pbokoc@redhat.com

Legal Notice

Abstract

Anaconda is the installer used by Red Hat Enterprise Linux, Fedora, and their derivatives. This document contains information necessary for customizing it. Developers who wish to expand the base functionality of the installer will find information about Anaconda architecture, its add-on API and provided helper functions, and examples which will help in creating custom add-ons. This guide also provides instructions for those wishing to customize visual aspects of the installer, such as the boot menu color scheme and background or branding and chroming within the graphical user interface.

Table of Contents

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 there are also more options for changing 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

Some of the topics discussed in this book require significant pre-existing knowledge. In particular, developing custom **Anaconda** add-ons requires knowledge of Python, making changes to the boot menu requires involves editing plain text configuration files, and visual customizations of the installer require familiarity with computer graphics and cascading style sheets (CSS).

Also note that this document only applies to Red Hat Enterprise Linux 7 and Fedora 17 and later.

Important

Procedures described in this book are written for Red Hat Enterprise Linux 7 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.

2. Working with ISO Images

This section will explain how to extract an ISO image provided by Red Hat, and how to create a new boot image containing changes you made following other procedures in this book.

2.1. Extracting Red Hat Enterprise Linux Boot Images

Before you start customizing the installer, you must download Red Hat-provided boot images. These images will be required to perform all procedures described in this book.

You can obtain Red Hat Enterprise Linux 7 boot media from the Red Hat Customer Portal after logging in to your account. Your account must have sufficient entitlements to download Red Hat Enterprise Linux 7 images.

Download either the **`Binary DVD`** or **`Boot ISO`** image from the Customer Portal. Either of these can be modified using procedures in this guide; other available downloads, such as the **`KVM Guest Image`** or **`Supplementary DVD`** can not. The *variant* of the image (such as **`Server`** or **`ComputeNode`**) does not matter in this case; any variant can be used.

For detailed download instructions and description of the Binary DVD and Boot ISO downloads, see the *Red Hat Enterprise Linux 7 Installation Guide*.

After your chosen iso image finishes downloading, follow the procedure below to extract its contents in order to prepare for their modification.

Procedure 1. Extracting ISO Images

1. Mount the downloaded image.

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

   Replace *path/to/image.iso* with the path to the downloaded ISO. Also make sure that the target directory (**/mnt/iso**) exists and nothing else is currently mounted there.

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

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

3. 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
   ```

4. Unmount the image.

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

After you finish unpacking, the ISO image is extracted in your **/tmp/ISO** where you can modify its contents. Continue with Section 3, “Customizing the Boot Menu” or Section 5, “Developing Installer Add-ons”. Once you finish making changes, create a new, modified ISO image using the instructions in Section 2.3, “Creating Custom Boot Images”.

2.2. Creating a product.img File

A **product.img** image file is an archive containing files which replace existing files or add new ones in the installer runtime. During boot, **Anaconda** loads this file from the **images/** directory on the boot media. Then, it uses files present inside this file to replace identically named files in the installer's file system; this is necessary to customize the installer (for example, for replacing default images with custom ones). The **product.img** image must contain a directory structure identical to the installer.

Specifically, two topics discussed in this guide require you to create a product image. The table below lists the correct locations inside the image file directory structure:

Table 1. Locations of Add-ons and Anaconda Visuals

Type of custom content	File system location
Pixmaps (logo, side bar, top bar, etc.)	**/usr/share/anaconda/pixmaps/**
Banners for the installation progress screen	**/usr/share/anaconda/pixmaps/rnotes/en/**
GUI stylesheet	**/usr/share/anaconda/anaconda-gtk.css**
Installclasses (for changing the product name)	**/run/install/product/pyanaconda/installclasses/**
Anaconda add-ons	**/usr/share/anaconda/addons/**

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

Procedure 2. Creating product.img

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

   ```
   $ cd /tmp
   ```

   ```
   $ mkdir product/
   ```

2. Create a directory structure which is identical to the location of the file you want to replace. For example, if you want to test an add-on, which belongs 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

   You can browse the installer's runtime file system by booting the installation, switching to virtual console 1 (**Ctrl**+**Alt**+**F1**) and then switching to the second **tmux** window (**Ctrl**+**b** **2**). This opens a shell prompt which you can use to browse the file system.

3. 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/
   ```

4. Repeat the two steps above (create a directory structure and move modified files into it) for every file you want to add to the installer.

5. Change into 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.

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

After finishing this procedure, your customizations are placed in the correct directory. You can continue with Section 2.3, “Creating Custom Boot Images” to create a new bootable ISO image with your changes included. The **`product.img`** file will be automatically loaded when starting the installer.

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.

See the *Red Hat Enterprise Linux 7 Installation Guide* for more information about **Anaconda** boot options.

2.3. Creating Custom Boot Images

When you finish customizing boot images provided by Red Hat, you must create a new image which includes changes you made. To do this, follow the procedure below.

Procedure 3. Creating ISO Images

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 - e.g. **`/tmp/ISO/iso`**.

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

```
# genisoimage -U -r -v -T -J -joliet-long -V "RHEL-7.1 Server.x86_64" -volset "RHEL-7.1 Server.x86_64" -A "RHEL-7.1 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 values for the **`-V`**, **`-volset`**, and **`-A`** options match the image's boot loader configuration, if you are using the **`LABEL=`** directive for options which 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 label of **`RHEL 7.1`**, boot loader configuration should use **`RHEL\x207.1`** to refer to this label.

- Replace the value of the **`-o`** option (**`-o ../NEWISO.iso`**) with the file name of your new image. The value in the example will create file **`NEWISO.iso`** 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. Without performing this step, image verification check (the **`rd.live.check`** option in the boot loader configuration) will fail and you will not be able to continue with the installation.

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

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

After finishing this procedure, you can 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. See the *Red Hat Enterprise Linux 7 Installation Guide* for instructions on preparing boot media or network server, and the *Red Hat Enterprise Linux 7 Virtualization Getting Started Guide* for instructions on creating virtual machines with ISO images.

3. Customizing the Boot Menu

This section contains information necessary for customizing the *boot menu* - the menu which appears after you boot your system from 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`**. These options can be customized and additional options can be added, and the visual style (colors and background) can be changed.

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

Customizing boot menu options can be especially 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 and adding the **`inst.ks=`** boot option; if you edit boot loader configuration files on the media, you can add this option to one of the pre-configured entries.

Before you begin customizing the boot loader, follow Procedure 1, “Extracting ISO Images” to unpack the ISC image you want to modify into a working directory. After you finish your modifications, follow Procedure 3, “Creating ISO Images” to create a new bootable ISO image.

3.1. Systems with BIOS Firmware

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

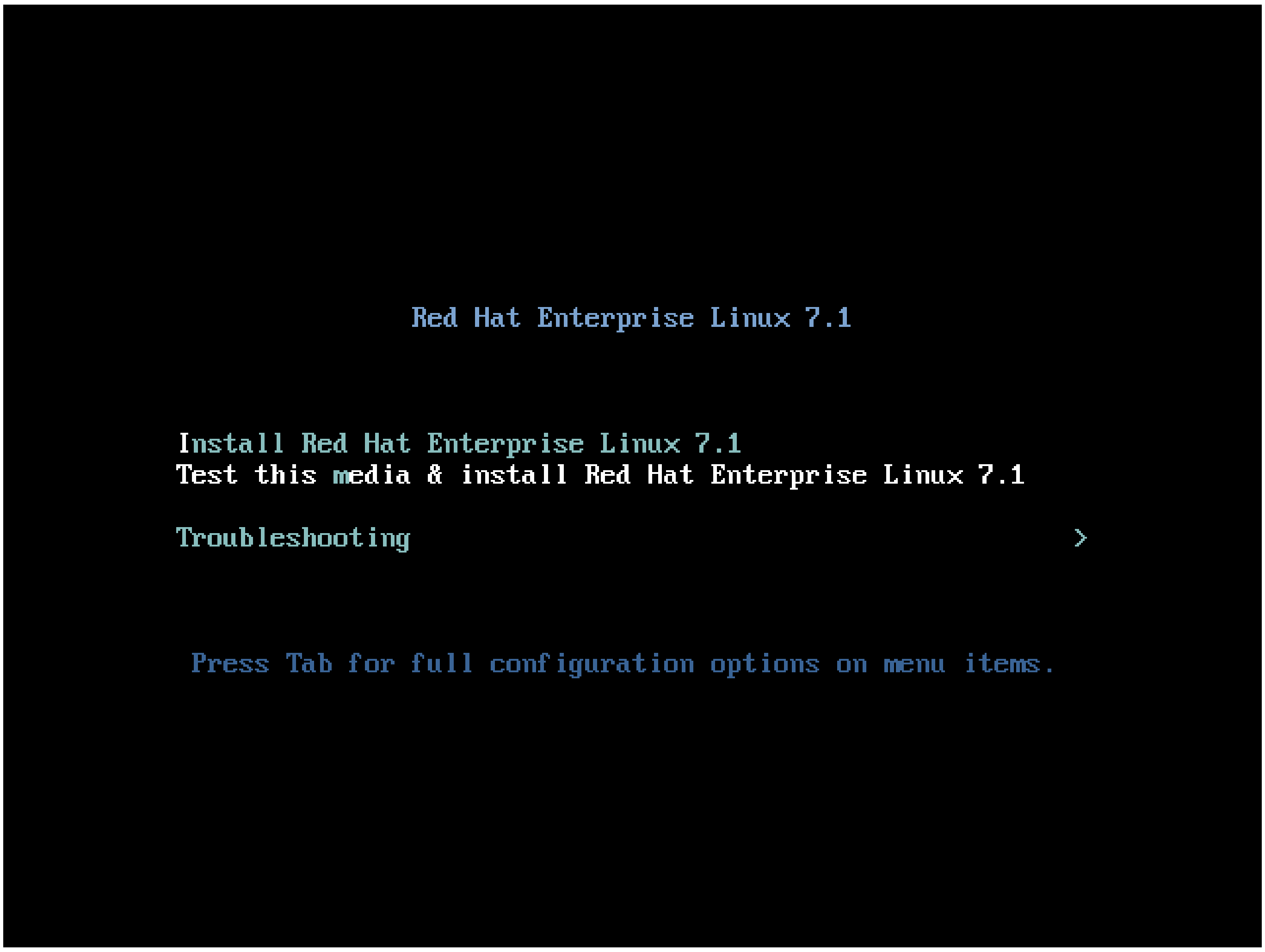
Red Hat Enterprise Linux 7.1
Install Red Hat Enterprise Linux 7.1
Test this media & install Red Hat Enterprise Linux 7.1
Troubleshooting >
Press Tab for full configuration options on menu items.

Figure 1. ISOLINUX Boot Menu

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

In the configuration file, the default menu entry for Red Hat Enterprise Linux, **`Test this media & Install Red Hat Enterprise Linux 7`**, is defined in the following block:

```
label check
  menu label Test this ^media & install Red Hat Enterprise Linux 7.1
  menu default
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64
rd.live.check quiet
```

Notable options in the above example are:

- **`menu label`** - determines how the entry will be named in the menu. The **`^`** character determines its keyboard shortcut (the **`m`** key).
- **`menu default`** - makes this option selected by default, 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.

Usable options which are specific to **Anaconda** are listed in the *Red Hat Enterprise Linux 7 Installation Guide*. One of the notable options is **`inst.ks=`**, which allows you to specify a location of a Kickstart file. Therefore, you can place a Kickstart file on the boot ISO image and use this option to use it; for example, you can place a file named **`kickstart.ks`** into the image's root directory and use **`inst.ks=hd:LABEL=RHEL-7.1\x20x86_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-7.1\x20x86_64`** option above), replace all spaces with **`\x20`**.

Other important options which are not part of menu entry definitions include:

- **`timeout`** - determines how long will the boot menu displayed before the default menu entry is automatically used. The default value it **`600`**, which means the menu will be displayed for 60 seconds. Setting this value to **`0`** disables the timeout completely.

Note

Setting the timeout to a low value such as **`1`** is useful when performing a headless installation as you will not have to wait for the 60 second 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) will look 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-7.1\x20x86_64 rescue
quiet

menu separator

label returntomain
  menu label Return to ^main menu
  menu exit

menu end
```

As you can see in the above example, 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 goes back 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 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) and *foreground* and *background* which determine the actual colors. Colors are described using an **`#AARRGGBB`** notation in hexadecimal format; the first pair of digits (*AA*) determines opacity (**`00`** for fully transparent, **`ff`** for fully opaque).

  See the Syslinux Wiki for detailed information about available elements, ANSI values, shadow settings and other visual customization options.

- **`menu help textfile`** - creates a menu entry which, when selected, displays a help text file.

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

3.2. Systems with UEFI Firmware

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

The **GRUB2** configuration file is **`EFI/BOOT/grub.cfg`** on the boot media. The configuration file contains a list of preconfigured menu entries and other directives which control the appearanace and functionality of the boot menu.

In the configuration file, the default menu entry for Red Hat Enterprise Linux (**`Test this media & install Red Hat Enterprise Linux 7.1`**) is defined in the following block:

```
menuentry 'Test this media & install Red Hat Enterprise Linux 7.1' --class
fedora --class gnu-linux --class gnu --class os {
  linuxefi /images/pxeboot/vmlinuz inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64
rd.live.check quiet
  initrdefi /images/pxeboot/initrd.img
}
```

Notable options in the above example are:

- **`menuentry`** - the option that defines the menu entry. The title of the entry is in single or double quotes (**`'`** or **`"`**). The **`--class`** option can be used to group menu entries into different *classes*, which can then be styled differently using **GRUB2** themes.

  Note

  Each menu entry definition must be enclosed in curly braces (**`{}`**) as shown in the above example.

- **`linuxefi`** - this option defines which kernel will be booted (**`/images/pxeboot/vmlinuz`** in the above example) as well as additional options. Customize these options to change the behavior of the boot entry.

  Usable options which are specific to **Anaconda** are listed in the *Red Hat Enterprise Linux 7 Installation Guide*. One of the notable options is **`inst.ks=`**, which allows you to specify a location of a Kickstart file. Therefore, you can place a Kickstart file on the boot ISO image and use this option to use it; for example, you can place a file named **`kickstart.ks`** into the image's root directory and use **`inst.ks=hd:LABEL=RHEL-7.1\x20x86_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-7.1\x20x86_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 will the boot menu displayed before the default menu entry is automatically used. The default value it **`60`**, which means the menu will be 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 will immediately activate the default boot entry.

- **`submenu`** - definition of a *submenu* block. This allows you to create a sub-menu and group some entries under it, instead of displaying them in the main menu. In the default configuration, there is a **`Troubleshooting`** submenu which 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-7.1\x20x86_64
xdriver=vesa nomodeset quiet
    initrdefi /images/pxeboot/initrd.img
  }
  menuentry 'Submenu option 2' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64
rescue quiet
    initrdefi /images/initrd.img
  }
}
```

- **`set default`** - this option determines which entry will be selected by default. Note that entry numbers start from **`0`**; if you want to make the *third* entry the default one, use **`set default=2`**, etc.

- **`theme`** - location of a directory which contains a **GRUB2** theme files. Themes can be used to customize visual aspects of the boot loader - background, fonts, and colors of specific elements.

Full description of the theme file format is beyond the scope of this document. For information about creating custom themes, see the GNU GRUB Manual 2.00.

For additional information about customizing the boot menu, see the GNU GRUB Manual 2.00. Also see the *Red Hat Enterprise Linux 7 System Administrator's Guide* for more general information about **GRUB2**.

4. Branding and Chroming the Graphical User Interface

The following sections describe changing the appearance of the graphical user interface (GUI) of the **Anaconda** installer.

There are several elements in the graphical user interface of **Anaconda** which can be changed to customize the look of the installer. To customize the installer's appearance, you must create a custom **`product.img`** file containing a custom *installclass* (to change the product name displayed in the installer) and your own branding material. The **`product.img`** file is not an installation image; it is used to supplement the full installation ISO image by loading your customizations and using them to overwrite files included on the boot image by default.

See Section 2, "Working with ISO Images" for information about extracting boot images provided by Red Hat, creating a **`product.img`** file and adding this file to the ISO images.

4.1. Customizing Graphical Elements

Graphical elements of the installer which can be changed are stored in the **`/usr/share/anaconda/pixmaps/`** directory in the installer runtime file system. This directory contains the following files:

```
pixmaps
├─ anaconda-selected-icon.svg
├─ dialog-warning-symbolic.svg
├─ right-arrow-icon.png
├─ rnotes
│  └─ en
│     ├─ RHEL_7_InstallerBanner_Andreas_750x120_11649367_1213jw.png
│     ├─ RHEL_7_InstallerBanner_Blog_750x120_11649367_1213jw.png
│     ├─
RHEL_7_InstallerBanner_CPAccess_CommandLine_750x120_11649367_1213jw.png
│     ├─ RHEL_7_InstallerBanner_CPAccess_Desktop_750x120_11649367_1213jw.png
│     ├─ RHEL_7_InstallerBanner_CPAccess_Help_750x120_11649367_1213jw.png
│     ├─ RHEL_7_InstallerBanner_Middleware_750x120_11649367_1213jw.png
│     ├─ RHEL_7_InstallerBanner_OPSEN_750x120_11649367_1213cd.png
│     ├─ RHEL_7_InstallerBanner_RHDev_Program_750x120_11649367_1213cd.png
│     ├─ RHEL_7_InstallerBanner_RHELStandardize_750x120_11649367_1213jw.png
│     └─ RHEL_7_InstallerBanner_Satellite_750x120_11649367_1213cd.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 side bar and top bar. The file has the following contents:

```
/* vendor-specific colors/images */
```

```
@define-color redhat #021519;

/* logo and sidebar classes for RHEL */

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

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

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

AnacondaSpokeWindow #layout-indicator {
    color: black;
}
```

The most imporant part of the CSS file is the way 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 style sheet 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.

The **`rnotes/`** directory contains a set of banners. During the installation, banner graphics cycle along the bottom of the screen, approximately once per minute.

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

Apart from graphical elements described in the previous section, you can also customize the product name displayed during the installation. This product name is shown in the top right corner in all screens.

To change the product name, you must create a custom *installation class*. Create a new file named **`custom.py`** with content similar to the example below:

Example 1. Creating a Custom Installclass

```
from pyanaconda.installclass import BaseInstallClass
from pyanaconda.product import productName
from pyanaconda import network
from pyanaconda import nm

class CustomBaseInstallClass(BaseInstallClass):
    name = "My Distribution"
    sortPriority = 30000
    if not productName.startswith("My Distribution"):
        hidden = True
    defaultFS = "xfs"
    bootloaderTimeoutDefault = 5
    bootloaderExtraArgs = []

    ignoredPackages = ["ntfsprogs"]

    installUpdates = False

    _l10n_domain = "comps"

    efi_dir = "redhat"

    help_placeholder = "RHEL7Placeholder.html"
    help_placeholder_with_links = "RHEL7PlaceholderWithLinks.html"

    def configure(self, anaconda):
        BaseInstallClass.configure(self, anaconda)
        BaseInstallClass.setDefaultPartitioning(self, anaconda.storage)

    def setNetworkOnbootDefault(self, ksdata):
        if ksdata.method.method not in ("url", "nfs"):
            return
        if network.has_some_wired_autoconnect_device():
            return
        dev = network.default_route_device()
        if not dev:
            return
        if nm.nm_device_type_is_wifi(dev):
            return
        network.update_onboot_value(dev, "yes", ksdata)

    def __init__(self):
        BaseInstallClass.__init__(self)
```

The file above determines the installer defaults (such as the default file system, etc.), but the part relevant to this procedure is the following block:

```
class CustomBaseInstallClass(BaseInstallClass):
    name = "My Distribution"
    sortPriority = 30000
    if not productName.startswith("My Distribution"):
        hidden = True
```

Change *My Distribution* to the name which you want to display in the installer. Also make sure that the **`sortPriority`** attribute is set to more than **`20000`**; this makes sure that the new installation class will be loaded first.

Warning

Do not change any other attributes or class names in the file - otherwise you may cause the installer to behave unpredictably.

After you create the custom installclass, follow the steps in Section 2.2, "Creating a product.img File" to create a new **`product.img`** file containing your customizations, and the Section 2.3, "Creating Custom Boot Images" to create a new bootable ISO file with your changes included.

5. Developing Installer Add-ons

5.1. Introduction to Anaconda and Add-ons

5.1.1. Introduction to Anaconda

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 such a system up on a machine. The installation process has four major steps:

- installation destination preparation (usually disk partitioning)
- package and data installation
- boot loader installation and configuration
- configuration of the newly installed system

There are three ways you can control the installer and specify installation options. The most common approach is to use the *graphical user interface* (GUI). This interface is meant to allow users to install the system interactively with little or no configuration required before beginning the installation, and it should cover all common use cases, including setting up complicated partitioning layouts.

The graphical interface also supports remote access over **`VNC`**, which allows you to use the GUI even on systems with no graphics cards or even attached monitor. However, there are still cases where this is not desired, but at the same time, you may want to perform an interactive installation. For these cases, a *text mode* (TUI) is available. The TUI works in a way similar to a monochrome line printer, which allows it to work even on serial consoles which do not support cursor movement, colors and other advanced features. The text mode is limited in that it only allows you to customize 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.

The third way to install a system using **Anaconda** is by using a Kickstart file - a plain text file with shell-like syntax which can contain data to drive the installation process. A Kickstart file allows you to partially or completely automate the installation. A certain set of commands which configures all required areas is necessary to completely automate the installation; if one or more of the required commands is missing, the installation will require interaction. If all required commands are present, the installation will be performed in a completely automatic way, without any need for interaction.

Kickstart provides the highest amount of options, covering use cases where neither the TUI nor the GUI is sufficient. Every feature in **Anaconda** must always be supported in Kickstart; other interfaces follow only subsets of all available options, which allows them to remain clear.

5.1.2. Firstboot and Initial Setup

The first boot of the newly installed system is traditionally considered a part of the installation process as well, because some parts of configuration such as user creation are often performed at this point. Previously, the **Firstboot** tool has been used for this purpose, allowing you to register your newly installer Red Hat Enterprise Linux system or configure **Kdump**. However, **Firstboot** relies on no longer maintained tools such as **`Gtk2`** and the **`pygtk2`** module. [1] For this reason, a new tool called **Initial Setup** was developed, which reuses code from **Anaconda**. This allows add-ons developed for **Anaconda** to be easily reused in **Initial Setup**. This topic is further discussed in Section 5.6, “Writing an Anaconda add-on”.

5.1.3. Anaconda and Initial Setup Add-ons

Installing a new operating system is a vastly complicated use case - each user may want to do something slightly different. Designing an installer for every corner case would cause it to be cluttered with rarely-used functionality. For this reason, when the installer was being rewritten into its current form, it gained support for add-ons.

Anaconda add-ons can be used to add your own Kickstart commands and options as well as new configuration screens in the graphical and text-based user interface, depending on your specific use case. Each add-on must have Kickstart support; the GUI and TUI are optional, but can be very helpful.

In current releases of Red Hat Enterprise Linux (7.1 and later) and Fedora [2] (21 and later), one add-on is included by default: The **Kdump** add-on, which adds support for configuring kernel crash dumping during the installation. This add-on has full support in Kickstart (using the **`%addon com_redhat_kdump`** command and its options) and is fully integrated as an additional screen in the text-based and graphical interfaces. You can develop other add-ons in the same way and add them to the default installer using procedures described further in this guide.

5.1.4. Additional Information

Following links contain additional information about **Anaconda** and **Initial Setup**:

- The Anaconda page on Fedora Project Wiki contains provides more information about the installer.
- Information about development of **Anaconda** into its current version is available at the Anaconda/NewInstaller Wiki page.
- The Kickstart Installations chapter of the *Red Hat Enterprise Linux 7 Installation Guide* provides full documentation of Kickstart, including a list of all supported commands and options.
- The Installing Using Anaconda chapter of the *Red Hat Enterprise Linux 7 Installation Guide* describes the installation process in the graphical and text user interfaces.
- For information about tools used for after-installation configuration, see Initial Setup and Firstboot.

5.2. Architecture of Anaconda

Anaconda is a set of Python modules and scripts. It also uses several external packages and libraries, some of which were created specifically for the installer. Major components of this toolset include the following packages:

- **`pykickstart`** - used to parse and validate Kickstart files and also to provide a data structure which stores values which drive the installation
- **`yum`** - the package manager which handles installation of packages and resolving dependencies
- **`blivet`** - originally split from the *anaconda* package as *pyanaconda.storage*; used to handle all activities related to storage management
- **`pyanaconda`** - package containing the core of the user interface and modules for functionality unique to **Anaconda**, such as keyboard and timezone selection, network configuration, and user creation, as well as a number of utilities and system-oriented functions
- **`python-meh`** - contains an exception handler which 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.

The life cycle of data during the installation process is straightforward. If a Kickstart file is provided, it is processed by the **`pykickstart`** module and imported into memory as a tree-like structure. If no Kickstart file is provided, an empty tree-like structure is created instead. If the installation is interactive (not all required Kickstart commands have been used), the structure is then updated with choices made by the user in the interactive interface.

Once all required choices are made, the installation process begins and values stored in the structure are used to determine parameters of the installation. The values are also written as a Kickstart file which is saved in the **`/root/`** directory on the installed system; therefore the installation can be replicated automatically by reusing this automatically generated Kickstart file.

Elements of the tree-like structure are defined by the *pykickstart* package, but some of them can be overriden by modified versions from the **`pyanaconda.kickstart`** module. An important rule which governs this behavior is that there is no place to store configuration data, and the installation process is data-driven and relies on transactions as much as possible. This enforces the following features:

- every feature of the installer *must* be supported in Kickstart
- there is a single, obvious point in the installation process where changes are written to the target system; before this point, no lasting changes (e.g. formatting storage) are made
- every change made manually in the user interface is reflected in the resulting Kickstart file and can be replicated

The fact that the installation is *data-driven* means that installation and configuration logic lies within the methods of the items in the tree-like structure. Every item is set up (the **`setup`** method) to modify the runtime environment of the installation if necessary, and then executed (the **`execute`** method) to perform the changes on the target system. These methods are further described in Section 5.6, “Writing an Anaconda add-on”.

5.3. The Hub & Spoke model

One of the notable differences between **Anaconda** and most other operating system installers is its non-linear nature, also known as the *hub and spoke* model.

The hub and spoke model of **Anaconda** has several advantages, including:

- users are not forced to go through the screens in some strictly defined order
- users are not forced to visit every screen no matter if they understand what the options configured in it mean or not

- it is good for the transactional mode where all desired values can be set while nothing is actually happening to the underlying machine until a special button is clicked
- it provides way to show an overview of the configured values
- it has a great support for extensibility, because additional spokes can be put on hubs without need to reorder anything and resolve some complex ordering dependencies
- it can be used for both graphical and text mode of the installer

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

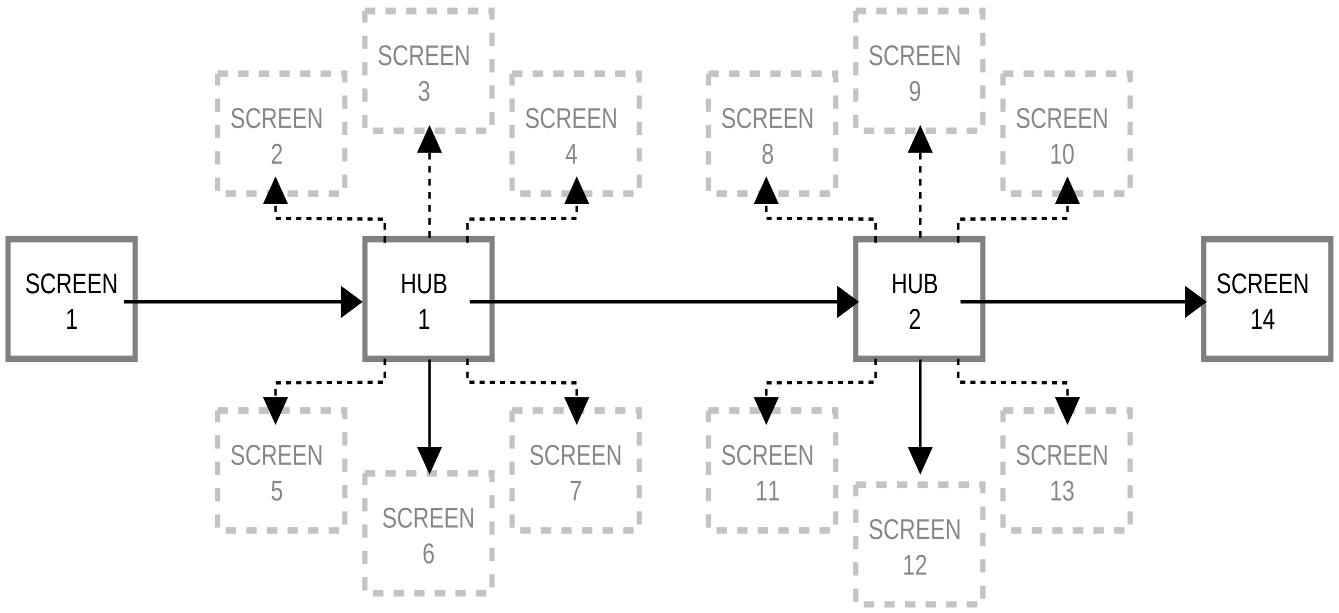

Figure 2. Diagram of the hub and spoke model

In the diagram, screens 2-13 are called *normal spokes*, and screens 1 and 14 are *standalone spokes*. Standalone spokes are a type of screen which is a type of screen that should be used only in case it has to be visited before (or after) the following (or previous) standalone spoke or hub. This may be, 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

Screens mentioned in the rest of this section are screens from the installer's graphical interface (GUI).

Central points of the hub and spoke model are *hubs*. There are two hubs by default:

- The **`Installation Summary`** hub which shows a summary of configured options before the installation begins
- The **`Configuration and Progress`** hub which appears after you click **`Begin Installation`** in **`Installation Summary`**, and which displays the progress of the installation process and allows you to configure additional options (set the root password and create a user account).

Each spoke has several predefined *properties* which are reflected on the hub. These are:

- ready - states whether the spoke can be visited or not; for example, when the installer is configuring a package source, that spoke is not ready, is colored gray, and cannot be accessed until configuration is complete
- completed - marks the spoke as completed (all required values are set) or not
- mandatory - determines whether the spoke *must* be visited and confirmed by the user before continuing the installation; for example, the **`Installation Destination`** spoke must always be visited, 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 sub-trees of the in-memory tree-like structure which was discussed in Section 5.2, “Architecture of Anaconda”. As Section 5.6, “Writing an Anaconda add-on” explains, the same applies to spokes provided by add-ons.

5.4. Threads and Communication

Some of the actions which need to be performed during the installation process, such as scanning disks for existing partitions or downloading package metadata, can take a long time. To prevent you from waiting and remain responsive if possible, **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 itself, and all code performing actions which involve the GUI must make sure that these actions are run in the main thread as well. The only supported way to do so is by using the **`GLib.idle_add`**, which is not always easy or desired. To alleviate this problem, several helper functions and decorators are defined in the *pyanaconda.ui.gui.utils* module.

The most useful of those are the **`@gtk_action_wait`** and **`@gtk_action_nowait`** decorators. They 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, run in the main thread, and the return value is either returned to the caller or dropped, respectively.

As mentioned previously, one of the main reasons for using multiple threads is to allow the user to configure some screens while other screens which are currently busy (such as **`Installation Source`** when it downloads package metadata) configure themselves. Once the configuration is finished, the spoke which was previously busy needs to announce that it is now ready and not blocked; this is handled by a message queue called **`hubQ`**, which is being periodically checked in the main event loop. When a spoke becomes accessible, it sends a message to this queue announcing this change and that it should no longer be blocked.

The same applies in a situation where a spoke needs to refresh its status or completion 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 needed for the text-based interface, where the situation is more complicated; there is no main loop in text mode, instead the majority of time in this mode is spent waiting for keyboard input.

5.5. Anaconda Add-on Structure

An **Anaconda** add-on is a Python package containing a directory with an **`__init__.py`** and other source directories (subpackages) inside. Because Python allows importing each package name only once, the package top-level directory name must be unique. At the same time, the name can be arbitrary, 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 therefore similar to Java packages or D-Bus service names: prefix the add-on name with the reversed domain name of your organization, using underscores (_) instead of dots so that the directory name is a valid identifier for a Python package. An example add-on name following these suggestions would therefore be e.g. **`com_example_hello_world`**. This convention follows the recommended naming scheme for Python package and module names.

Important

Make sure to create an **`__init__.py`** file in each directory. Directories missing this file are not considered valid Python packages.

When writing an add-on, keep in mind that every function supported in the installer *must* be supported in Kickstart; GUI and TUI support is optional. Support for each interface (Kickstart, graphical interface and text interface) must be in a separate subpackage and these subpackages must be named **`ks`** for Kickstart, **`gui`** for the graphical interface and **`tui`** for the text-based interface. The **`gui`** and **`tui`** packages must also contain a **`spokes`** subpackage. [3]

Names of modules inside these packages are arbitrary; the **`ks/`**, **`gui/`** and **`tui/`** directories can contain Python modules with any name.

A sample directory structure for an add-on which supports every interface (Kickstart, GUI and TUI) will look similar to the following:

Example 2. Sample Add-on Structure

```
com_example_hello_world
├─ ks
│   └─ __init__.py
├─ 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 classes inherited from one or more classes defined in the API. This is further discussed in Section 5.6, “Writing an Anaconda add-on”.

All add-ons should 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 automatically generated documentation for your add-on, docstrings should follow the guidelines for the toolkit you use to accomplish this.

5.6. Writing an Anaconda add-on

The sections below will demonstrate the process writing and testing a sample add-on called Hello World. This sample add-on will support all interfaces (Kickstart, GUI and TUI). Sources for this sample add-on are available on GitHub in the rhinstaller/hello-world-anaconda-addon repository; it is recommended to clone this repository or at least open the sources in the web interface.

Another repository to review is rhinstaller/anaconda, which contains the installer source code; it will be referred to in several parts of this section as well.

Before you begin developing the add-on itself, start by creating its directory structure as described in Section 5.5, "Anaconda Add-on Structure". Then, continue with Section 5.6.1, "Kickstart Support", as Kickstart support is mandatory for all add-ons. After that, you can optionally continue with Section 5.6.2, "Graphical user interface" and Section 5.6.3, "Text User Interface" if needed.

5.6.1. Kickstart Support

Kickstart support is always the first part of any add-on that should be developed. Other packages - support for the graphical and text-based interface - will depend on it. To begin, navigate to the **`com_example_hello_world/ks/`** directory you have created previously, make sure it contains an **`__init__.py`** file, and add another Python script named **`hello_world.py`**.

Unlike built-in Kickstart commands, add-ons are used in their own *sections*. Each use of an add-on in a Kickstart file begins with an **`%addon`** statement and is closed by **`%end`**. The **`%addon`** line also contains the name of the add-on (such as **`%addon com_example_hello_world`**) and optionally a list of arguments, if the add-on supports them.

An example use of an add-on in a Kickstart file looks like the example below:

Example 3. Using an Add-on in a Kickstart File

```
%addon ADDON_NAME [arguments]
first line
second line
...
%end
```

The key class for Kickstart support in add-ons is called **`AddonData`**. This class is defined in *pyanaconda.addons* and represents an object for parsing and storing data from a Kickstart file.

Arguments are passed as a list to an instance of the add-on class inherited from the **`AddonData`** class. Anything between the first and last line is passed to the add-on's class one line at a time. To keep the example Hello World add-on simple, it will merge all lines in this block into a single line and separate the original lines with a space.

The example add-on requires a class inherited from **`AddonData`** with a method for handling the list of arguments from the **`%addon`** line, and a method for handling lines inside the section. The **`pyanaconda/addons.py`** module contains two methods which can be used for this:

- **`handle_header`** - takes a list of arguments from the **`%addon`** line (and line numbers for error reporting)
- **`handle_line`** - takes a single line of content from between the **`%addon`** and **`%end`** statements

The example below demonstrates a Hello World add-on which uses the methods described above:

Example 4. Using handle_header and handle_line

```
from pyanaconda.addons import AddonData
from pykickstart.options import KSOptionParser

# export HelloWorldData class to prevent Anaconda's collect method from
taking
# AddonData class instead of the HelloWorldData class
# :see: pyanaconda.kickstart.AnacondaKSHandler.__init__
__all__ = ["HelloWorldData"]

HELLO_FILE_PATH = "/root/hello_world_addon_output.txt"

class HelloWorldData(AddonData):
    """
    Class parsing and storing data for the Hello world addon.

    :see: pyanaconda.addons.AddonData

    """

def __init__(self, name):
    """
    :param name: name of the addon
    :type name: str

    """

    AddonData.__init__(self, name)
    self.text = ""
    self.reverse = False

def handle_header(self, lineno, args):
    """
    The handle_header method is called to parse additional arguments in
the
    %addon section line.

    :param lineno: the current linenumber in the kickstart file
    :type lineno: int
    :param args: any additional arguments after %addon <name>
    :type args: list

    """

    op = KSOptionParser()
    op.add_option("--reverse", action="store_true", default=False,
    dest="reverse", help="Reverse the display of the addon text")
    (opts, extra) = op.parse_args(args=args, lineno=lineno)

    # Reject any additoinal arguments. Since AddonData.handle_header
    # rejects any arguments, we can use it to create an error message
    # and raise an exception.
    if extra:
    AddonData.handle_header(self, lineno, extra)

    # Store the result of the option parsing
    self.reverse = opts.reverse
```

```
def handle_line(self, line):
    """
    The handle_line method that is called with every line from this
addon's
    %addon section of the kickstart file.

    :param line: a single line from the %addon section
    :type line: str

    """

    # simple example, we just append lines to the text attribute
    if self.text is "":
    self.text = line.strip()
    else:
    self.text += " " + line.strip()
```

The example begins by importing necessary methods and defining an `__all__` variable which is necessary to prevent **Anaconda**'s collect method from taking the **`AddonData`** class instead of add-on specific **`HelloWorldData`**.

Then, the example shows a definition of the **`HelloWorldData`** class inherited from **`AddonData`** with its `__init__` method calling the parent's `__init__` and initializing the attributes **`self.text`** and **`self.reverse`** to **`False`**.

The **`self.reverse`** attribute is populated in the **`handle_header`** method, and the **`self.text`** is populated in **`handle_line`**. The **`handle_header`** method uses an instance of the **`KSOptionParser`** provided by **`pykickstart`** to parse additional options used on the **`%addon`** line, and **`handle_line`** strips the content lines of white space at the beginning and end of each line, and appends them to **`self.text`**.

The code above covers the first phase of the data life cycle in the installation process: it reads data from the Kickstart file. The next step is to use this data to drive the installation process. Two predefined methods are available for this purpose:

- **`setup`** - called before the installation transaction starts and used to make changes to the installation runtime environment
- **`execute`** - called at the end of the transaction and used to make changes to the target system

To use these two methods, you must add some new imports and a constant to your module, as shown in the following example:

Example 5. Importing the setup and execute Methods

```
import os.path

from pyanaconda.addons import AddonData
from pyanaconda.constants import ROOT_PATH

HELLO_FILE_PATH = "/root/hello_world_addon_output.txt"
```

An updated example of the Hello World add-ons with the **`setup`** and **`execute`** methods included is below:

Example 6. Using the setup and execute Methods

```
def setup(self, storage, ksdata, instclass):
    """
    The setup method that should make changes to the runtime environment
    according to the data stored in this object.

    :param storage: object storing storage-related information
    (disks, partitioning, bootloader, etc.)
    :type storage: blivet.Blivet instance
    :param ksdata: data parsed from the kickstart file and set in the
    installation process
    :type ksdata: pykickstart.base.BaseHandler instance
    :param instclass: distribution-specific information
    :type instclass: pyanaconda.installclass.BaseInstallClass

    """

    # no actions needed in this addon
    pass

def execute(self, storage, ksdata, instclass, users):
    """
    The execute method that should make changes to the installed system.
It
    is called only once in the post-install setup phase.

    :see: setup
    :param users: information about created users
    :type users: pyanaconda.users.Users instance

    """

    hello_file_path = os.path.normpath(ROOT_PATH + HELLO_FILE_PATH)
    with open(hello_file_path, "w") as fobj:
        fobj.write("%s\n" % self.text)
```

In the above example, the **`setup`** method does nothing; the Hello World add-on does not make any changes to the installation runtime environment. The **`execute`** method writes stored text into a file created in the target system's root (**`/`**) directory.

The most important information in the above example is the amount and meaning of the arguments passed to the two new methods; these are described in docstrings within the example.

The final phase of the data life cycle, as well as the last part of the code needed in a module providing Kickstart support, is generating a new Kickstart file, which includes values set at installation time, at the end of the installation process as described in Section 5.2, “Architecture of Anaconda”. This is performed by calling the __**`str`**__ method recursively on the tree-like structure storing installation data, which means that the class inherited from **`AddonData`** must define its own __**`str`**__ method which returns its stored data in valid Kickstart syntax. This returned data must be possible to parse again using **`pykickstart`**.

In the Hello World example, the __**`str`**__ method will be similar to the following example:

Example 7. Defining a __str__ Method

```
def __str__(self):
    """
    What should end up in the resulting kickstart file, i.e. the %addon
    section containing string representation of the stored data.

    """

    addon_str = "%%addon %s" % self.name

    if self.reverse:
    addon_str += "--reverse"

    addon_str += "\n%s\n%%end" % self.text
    return addon_str
```

Once your Kickstart support module contains all necessary methods (**`handle_header`**, **`handle_line`**, **`setup`**, **`execute`** and **`__str__`**), it becomes a valid **Anaconda** add-on. You can continue with the following sections to add support for the graphical and text-based user interfaces, or you can continue with Section 5.7, “Deploying and testing an Anaconda add-on” and test the add-on.

5.6.2. Graphical user interface

This section will describe adding support for the graphical user interface (GUI) to your add-on. Before you begin, make sure that your add-on already includes support for Kickstart as described in the previous section.

Note

Before you start developing add-ons with support for the graphical interface, make sure to install the *anaconda-widgets* and *anaconda-widgets-devel* packages, which contain Gtk widgets specific for **Anaconda** such as **`SpokeWindow`**.

5.6.2.1. Basic features

Similarly to Kickstart support in add-ons, GUI support requires every part of the add-on to contain at least one module with a definition of a class inherited from a particular class defined by the API. In case of graphical support, the only recommended class is **`NormalSpoke`**, which is defined in **`pyanaconda.ui.gui.spokes`**. As the class name suggests, it is a class for the *normal spoke* type of screen as described in Section 5.3, “The Hub & Spoke model”.

To implement a new class inherited from **`NormalSpoke`**, you must define the following class attributes which are required by the API:

- **`builderObjects`** - lists all top-level objects from the spoke's **`.glade`** file that should be, with their children objects (recursively), exposed to the spoke - or should be an empty list if everything should be exposed to the spoke (not recommended)
- **`mainWidgetName`** - contains the id of the main window widget [4] 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

Example module with all required definitions is shown in the following example:

Example 8. 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(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.

    :see: pyanaconda.ui.common.UIObject
    :see: pyanaconda.ui.common.Spoke
    :see: pyanaconda.ui.gui.GUIObject

    """

    ### 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 is used to export the spoke class, followed by the first lines of its definition including definitions of attributes mentioned above. The values of these attributes are referencing widgets defined in `com_example_hello_world/gui/spokes/hello.glade` file.

Two other notable attributes are present. The first is **`category`**, which has its value imported from the **`HelloWorldCategory`** class from the `com_example_hello_world.gui.categories` module. The **`HelloWorldCategory`** class will be discussed later, but for now, note that the path to add-ons is in sys.path so that things can be imported from the *com_example_hello_world* package.

The second notable attribute in the example is **`title`**, which contains two underscores in its definition. The first one is part of the **N_** function name which marks the string for translation, but returns the non-translated version of the string (translation is done later). The second underscore marks the beginning of the title itself and makes the spoke reachable 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 for two such functions is that the GUI objects may be created in memory at one time and fully initialized (which can take a longer time) at a different time. 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 sample Hello World add-on, these two methods are defined as follows (note the number and description of the arguments passed to the __**init**__ method):

Example 9. Defining the __init__ and initialize Methods

```
def __init__(self, data, storage, payload, instclass):
    """
    :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
    :param instclass: distribution-specific information
    :type instclass: pyanaconda.installclass.BaseInstallClass

    """

    NormalSpoke.__init__(self, data, storage, payload, instclass)

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

        """

        NormalSpoke.initialize(self)
        self._entry = self.builder.get_object("textEntry")
```

Note the **data** parameter passed to the **__init__** method. This 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.

Because the **HelloWorldData** class has already been defined in Section 5.6.1, “Kickstart Support”, there already is a subtree in **self.data** for this add-on, and its root (an instance of the class) is available as **self.data.addons.com_example_hello_world**.

One of the other things an ancestor's **__init__** does is initializing an instance of the **GtkBuilder** with the spoke's **.glade** file and storing it as **self.builder**. This is used in the **initialize** method 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 an user who wants to change or review the values this spoke 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 make sure 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 the spoke is left 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 10. 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
    self.data.

    :see: pyanaconda.ui.common.UIObject.refresh

    """

    self._entry.set_text(self.data.addons.org_fedora_hello_world.text)

def apply(self):
    """
    The apply method that is called when the spoke is left. It should
    update the contents of self.data with values set in the GUI elements.
```

```
        """

        self.data.addons.org_fedora_hello_world.text = self._entry.get_text()

    def execute(self):
      """
      The excecute method that is called when the spoke is left. 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 (e.g. 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 (e.g. the **`Installation Destination`** spoke, which must be always 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 some value to be set in the **`text`** attribute of the **`HelloWorldData`** class:

Example 11. Defining the ready, completed and mandatory Methods

```
@property
def ready(self):
    """
    The ready property that tells whether the spoke is ready (can be
visited)
    or not. The spoke is made (in)sensitive based on the returned value.

    :rtype: bool

    """

    # this spoke is always ready
    return True

@property
def completed(self):
    """
    The completed property that tells whether all mandatory items on the
    spoke are set, or not. The spoke will be marked on the hub as
completed
    or uncompleted acording to the returned value.

    :rtype: bool
    """

    return bool(self.data.addons.org_fedora_hello_world.text)

@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 defining these properties, 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 12. 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

    """

    text = self.data.addons.org_fedora_hello_world.text

    # If --reverse was specified in the kickstart, reverse the text
    if self.data.addons.org_fedora_hello_world.reverse:
        text = text[::-1]

    if text:
        return _("Text set: %s") % text
    else:
    return _("Text not set")
```

After defining all properties described in this chapter, the add-on has full support for the graphical user interface as well as Kickstart. Note that 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 some other widgets specific to **Anaconda**, is found in the *anaconda-widgets* package. Other files required for development of add-ons with GUI support (such as Glade definitions) can be found 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 Section 5.7, "Deploying and testing an Anaconda add-on" and test the add-on.

5.6.2.2. Advanced features

The **`pyanaconda`** package contains several helper and utility functions and constructs which may be used by hubs and spokes and which have not been covered in the previous section. Most of them are located in **`pyanaconda.ui.gui.utils`**.

The sample Hello World add-on demonstrates usage of the **`englightbox`** content manager which is also used in **Anaconda**. This manager can put a window into a lightbox to increase its visibility and focus it and 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 which 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 13. Defining a englightbox Dialog

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

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

The code above creates an instance of the dialog and then uses the **`enlightbox`** context manager to run the dialog within a lightbox. The context manager needs a reference to the window of the spoke and to the dialog's window to instantiate the lightbox for them.

Another useful feature provided by **Anaconda** is the ability to define a spoke which will appear both during the installation and after the first reboot (in the **Initial Setup** utility described in Section 5.1.2, "Firstboot and Initial Setup"). To make a spoke available in both **Anaconda** and **Initial Setup**, you must inherit the special **`FirstbootSpokeMixIn`** (or, more precisely, 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**, you should instead inherit the **`FirstbootOnlySpokeMixIn`** class.

There are many more advanced features provided by the *pyanaconda* package (like the **`@gtk_action_wait`** and **`@gtk_action_nowait`** decorators), but they are out of scope of this guide. Readers are recommended to go through the installer's sources for examples.

5.6.3. Text User Interface

The third supported interface, after Kickstart and GUI which have been discussed in previous sections, **Anaconda** also supports a text-based interface. This interface is more limited in its capabilities, but on some systems it may be the only choice for an interactive installation. For more information about differences between the text-based and graphical interface and about limitations of the TUI, see Section 5.1.1, "Introduction to Anaconda".

To add support for the text interface into your add-on, create a new set of subpackages under the **`tui`** directory as described in Section 5.5, "Anaconda Add-on Structure".

Text mode support in the installer is based on the **`simpleline`** utility, which only allows very simple user interaction. It does *not* support cursor movement (instead acting like a line printer) nor any visual enhancements like using different colors or fonts.

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

For an add-on, the most important classes are **`NormalTUISpoke`** and various other classes defined in the **`pyanaconda.ui.tui.spokes`** package. All of those classes are based on the **`TUIObject`** class, which itself is an equivalent of the **`GUIObject`** class discussed in the previous chapter. 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, same 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

Categories are handled differently than in the GUI. [5] 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`**, and **`prompt`**, and properties (**`ready`**, **`completed`**, **`mandatory`**, and **`status`**). All of these have already been described in Section 5.6.2, "Graphical user interface".

The example below shows the implementation of a simple TUI spoke in the Hello World sample add-on:

Example 14. Defining a Simple TUI Spoke

```
def __init__(self, app, data, storage, payload, instclass):
    """
    :see: pyanaconda.ui.tui.base.UIScreen
    :see: pyanaconda.ui.tui.base.App
    :param app: reference to application which is a main class for TUI
                screen handling, it is responsible for mainloop control
                and keeping track of the stack where all TUI screens are
                scheduled
```

```
        :type app: instance of pyanaconda.ui.tui.base.App
        :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
        :param instclass: distribution-specific information
        :type instclass: pyanaconda.installclass.BaseInstallClass

        """

        NormalTUISpoke.__init__(self, app, data, storage, payload, instclass)
        self._entered_text = ""

    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

        """

        NormalTUISpoke.initialize(self)

    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: pyanaconda.ui.tui.base.UIScreen.refresh
        :param args: optional argument that may be used when the screen is
                     scheduled (passed to App.switch_screen* methods)
        :type args: anything
        :return: whether this screen requests input or not
        :rtype: bool

        """

        self._entered_text = self.data.addons.org_fedora_hello_world.text
        return True

    def apply(self):
        """
        The apply method that is called when the spoke is left. It should
        update the contents of self.data with values set in the spoke.

        """

        self.data.addons.org_fedora_hello_world.text = self._entered_text
```

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

    """

    # nothing to do here
    pass

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

    :param args: optional argument that may be used when the screen is
                 scheduled (passed to App.switch_screen* methods)
    :type args: anything
    :param key: user's input
    :type key: unicode
    :return: if the input should not be handled here, return it, otherwise
             return True or False if the input was processed succesfully
or
             not respectively
    :rtype: bool|unicode

    """

    if key:
    self._entered_text = key

    # no other actions scheduled, apply changes
    self.apply()

    # close the current screen (remove it from the stack)
    self.close()
    return True

def prompt(self, args=None):
    """
    The prompt method that is called by the main loop to get the prompt
    for this screen.

    :param args: optional argument that can be passed to
App.switch_screen*
                 methods
    :type args: anything
    :return: text that should be used in the prompt for the input
    :rtype: unicode|None

    """

    return _("Enter a new text or leave empty to use the old one: ")
```

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.

The **`initialize`** method sets up a default value for the internal attribute of the spoke, which is then updated by the **`refresh`** method and used by the **`apply`** method to update Kickstart data. The only differences in these two methods from their equivalents in the GUI is the return type of the **`refresh`** method (bool instead of None) and an additional **`args`** argument they take. The meaning of the returned value is explained in the comments - it tells the application (the **`App`** class instance) whether this spoke requires user input or not. The additional **`args`** argument is used for passing extra information to the spoke when scheduled.

The **`execute`** method has the same purpose as the equivalent method in the GUI; in this case, the method does nothing.

Methods **`input`** and **`prompt`** are specific to the text interface; there are no equivalents in Kickstart or GUI. These two methods are responsible for user interaction.

The **`prompt`** method should return a prompt which will be displayed after the content of the spoke is printed. After a string is entered in reaction to the prompt, this string is passed to the **`input`** method for processing. The **`input`** method then 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 complicated add-ons, you typically need to perform some non-trivial actions, such as parse **`c`** as "continue" or **`r`** as "refresh", convert numbers into integers, show additional screens or toggle boolean values.

Return value of the **`input`** class must be either the **`INPUT_PROCESSED`** or **`INPUT_DISCARDED`** constant (both of these are defined in the `pyanaconda.constants_text` module), *or* the input string itself (in case this input should be processed by a different screen).

In contrast to the graphical mode, the **`apply`** method is not called automatically when leaving the spoke; it must be called explicitly from the **`input`** method. The same applies to closing (hiding) the spoke's screen, which is done by calling the **`close`** method.

To show another screen (for example, if you need additional information which was entered in a different spoke), you can instantiate another **`TUIObject`** and call one of the **`self.app.switch_screen*`** methods of the **`App`**.

Due to restrictions of the text-based interface, TUI spokes tend to have a very similar structure: a list of checkboxes or entries which should be checked or unchecked and populated by the user. The previous paragraphs show a way to implement a TUI spoke where the its methods handle printing and processing of the available and provided data. However, there is a different way to accomplish this using the **`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 which should be set in it. The example below demonstrates this:

Example 15. Using EditTUISpoke to Define a Text Interface Spoke

```
class _EditData(object):
    """Auxiliary class for storing data from the example EditSpoke"""

    def __init__(self):
        """Trivial constructor just defining the fields that will store
data"""

        self.checked = False
        self.shown_input = ""
        self.hidden_input = ""

class HelloWorldEditSpoke(EditTUISpoke):
    """Example class demonstrating usage of EditTUISpoke inheritance"""

    title = _("Hello World Edit")
    category = "localization"

    # simple RE used to specify we only accept a single word as a valid
input
    _valid_input = re.compile(r'\w+')

    # special class attribute defining spoke's entries as:
    # Entry(TITLE, ATTRIBUTE, CHECKING_RE or TYPE, SHOW_FUNC or SHOW)
    # where:
    #   TITLE specifies descriptive title of the entry
    #   ATTRIBUTE specifies attribute of self.args that should be set to
the
    #             value entered by the user (may contain dots, i.e. may
specify
    #             a deep attribute)
    #   CHECKING_RE specifies compiled RE used for deciding about
    #               accepting/rejecting user's input
    #   TYPE may be one of EditTUISpoke.CHECK or EditTUISpoke.PASSWORD
used
    #        instead of CHECKING_RE for simple checkboxes or password
entries,
    #        respectively
    #   SHOW_FUNC is a function taking self and self.args and returning
True or
    #             False indicating whether the entry should be shown or
not
    #   SHOW is a boolean value that may be used instead of the SHOW_FUNC
    #
    #   :see: pyanaconda.ui.tui.spokes.EditTUISpoke
    edit_fields = [
        Entry("Simple checkbox", "checked", EditTUISpoke.CHECK, True),
        Entry("Always shown input", "shown_input", _valid_input, True),
        Entry("Conditioned input", "hidden_input", _valid_input,
        lambda self, args: bool(args.shown_input)),
        ]

    def __init__(self, app, data, storage, payload, instclass):
        EditTUISpoke.__init__(self, app, data, storage, payload,
instclass)

        # just populate the self.args attribute to have a store for data
        # typically self.data or a subtree of self.data is used as
self.args
        self.args = _EditData()

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

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

The auxiliary class **_EditData** serves as a data container which is used to store values entered by the user. The **HelloWorldEditSpoke** class defines a simple spoke with one checkbox and two entries, all of which are instances of the **EditTUISpokeEntry** class imported as the **Entry** class). The first one is shown every time the spoke is displayed, the second instance is only shown if the first one contains a non-empty value.

For more information about the **EditTUISpoke** class, see the comments in the above example.

5.7. Deploying and testing an Anaconda add-on

To test a new add-on, you must load it into the installation environment. Add-ons are collected from the ***/usr/share/anaconda/addons/*** directory in the installation runtime environment; to add your own add-on into that directory, you must create a **product.img** file with the same directory structure and place it on your boot media.

For specific instructions on unpacking an existing boot image, creating a **product.img** file and repackaging the image, see Section 2, “Working with ISO Images”.

A. Revision History

Revision	Date	Author
Revision 2-2	**Mon Nov 16 2015**	**Petr Bokoč**
Version for Red Hat Enterprise Linux 7.2 GA release		
Revision 2-1	**Mon Jun 22 2015**	**Petr Bokoč**
Finished editing the Anaconda Add-on Development section Added a section on customizing Anaconda visuals		
Revision 2-0	**Fri Jun 19 2015**	**Petr Bokoč**
Changed book title to Anaconda Customization Guide Added section about unpacking and repacking ISO images Added section about customizing boot menu Edited most of the Anaconda Add-on Development section for better readability		
Revision 1-0	**Wed Jan 15 2014**	**Vratislav Podzimek**
First version to be a part of the official documentation		
Revision 0-0	**Fri Dec 28 2012**	**Vratislav Podzimek**
Initial creation of book by publican		

Index

[1] While Firstboot is a legacy tool, it is still supported because of third-party modules written for it.

[2] In Fedora, the add-on is disabled by default. You can enable it using the **`inst.kdump_addon=on`** option in the boot menu.

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

[4] an instance of the SpokeWindow widget which is a custom widget created for the Anaconda installer

[5] which is likely to change in the future to sticking to the better (GUI) way