rPath Appliance Platform Agent Plugin Development Guide

3.0.2.1-2008-09-11

rPath Appliance Platform Agent Plugin Development Guide: 3.0.2.1-2008-09-11

Copyright © 2008 rPath, Inc.

rPath, Inc.
701 Corporate Center Drive, Suite 450
Raleigh, North Carolina 27607
USA

rPath, rBuilder, rPath Appliance Platform, and Conary are registered trademarks of rPath, Inc. All other trademarks and service marks are property of their respective owners.

1. Introduction 7
    What You Should Already Know 7
2. rAPA Plugin Development Process 9
3. Plugin Development Environment 11
    Supporting Software Setup 11
    rAPA Source Checkout Setup 13
    Plugin Development Directory Setup 14
    Generate Test Certificates 15
    Plugin Generator Setup 16
    Run rAPA in Development Mode 16
4. Generate a Template Plugin 19
5. Task Interface Layout 21
    JavaScript Handling 21
    View the Template Layout 21
    Set Up Task Headings and Text 22
    Form Elements 25
        Text Entry Field 26
        Drop-down List 26
        Highlight Selection List 28
        Scheduler Tool and Radio Buttons 29
        Checkbox Selection 31
        Date and Time Selection 33
        Status and Dialog Boxes 33
    Tabbed Interface Layout 33
6. Function Development for the Web Part 37
7. Function Development for the Server Part 39
8. Expose Plugin Functions for the XML-RPC Interface 41
9. Add Task Help 43
10. Package Custom Plugins for rAPA 45

Chapter 1. Introduction

rPath offers its customers products and services to build appliances based on the rPath Appliance Platform. Part of the platform is the rPath Appliance Platform Agent (rAPA) which provides an interface for performing initial configuration and ongoing maintenance on the deployed appliance. rAPA is made up of a server and exposed interfaces, and there are options to use either its native web interface or its XML-RPC calls to perform the administrative tasks.

One of the most powerful features of rAPA is the ability to customize it and extend it so that it provides a complete administrative interface for all appliance configuration and maintenance needs. rPath provides two documents for working with rAPA:

- *rAPA Customization Guide [http://docs.rpath.com/rAPA_Customization_Guide/index.html]* -- This guide introduces the default rAPA interface and provides instructions for packaging a custom configuration to brand the interface, modify the available functions, and override default values. This includes ensuring appliances that are using the rPath Appliance Platform Entitlement Service (rES) have the `entitlements.xml` file packaged and included as part of the appliance.
- *rPath Appliance Platform Agent Plugin Development Guide* -- This guide describes the structure of rAPA with an emphasis on its extensible nature, and it provides instructions for developing a custom plugin from a basic template and packaging that plugin to be part of an appliance.

What You Should Already Know

This guide assumes the reader has some basic appliance development and Conary packaging experience. If you are using this guide for the first time, and have not yet worked on your own appliance development with rPath products and services, rPath recommends either attending training classes, or just starting out with the self-paced *Application to Appliance: A Hands-on Guide*, including an optional pre-configured development environment, available online and as a PDF download linked from the following URL:

http://wiki.rpath.com/wiki/Application_to_Appliance

In addition, this guide assumes that the reader has referenced the *rPath Appliance Platform Agent CUSTOMIZATIONGUIDE;* to customize rAPA and its built-in tasks. This guide will reference the information in the customization guide, and readers are encouraged to use it as a supplement, either online or in PDF form, from the following URLs:

http://docs.rpath.com/rAPA_Customization_Guide (HTML)

http://docs.rpath.com/rAPA_Customization_Guide.pdf

Besides rPath documentation, appliance developers who are programming one or more plugins to extend rAPA should find a Python programming resource for reference. Experienced programmers who have not worked with Python may find *Dive Into Python* (www.diveintopython.org [http://www.diveintopython.org/]) a valuable tool to understanding how Python differs from other languages. Python-based technologies used in rAPA, such as Kid and CherryPy, are described more within the guide, including links to online resources for them.

Chapter 2. rAPA Plugin Development Process

Plugin development for rAPA is approached differently by different developers, but rPath has some guidelines that developers can follow to get started:

- Plan to make the operations of each custom plugin part of the quality assurance (QA) process for the appliance. Ensure that the plugins work as desired with the appliance as a whole, whether the appliance uses the customer-supported rPath Appliance Platform and rAPA, or whether it uses community-supported components from rPath Linux and the rBuilder Online version of rAPA.
- Select a version of rAPA that will be used on the appliance, and establish a plan to test and updated custom plugins with updated code to take advantage of rAPA fixes and features in future rAPA releases.
- Deploy a test appliance that can be used to test code as it is developed. This could be a basic appliance group with no more than rAPA in its manifest, or it could be a complete appliance that includes rAPA. Be sure the test appliance has all the software required to perform whatever action the rAPA plugin is developed to perform.
- Set up your appliance development environment so that it has all the software and configuration required for plugin development work. See Chapter 3, *Plugin Development Environment* for instructions on setting up this environment.
- For plugins that have a web interface, work on the visual layout of the plugin (web part) before programming the back-end (server part). This guide steps through development using this approach.
- For continued reference about existing plugins and how they implement their functions, including functions accessible using XML-RPC, reference the *Standard Plugins* section of the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html].
- Package the completed plugin and test its install and function on a test appliance with the corresponding version of rAPA for which it was developed. Use the information in Chapter 10, *Package Custom Plugins for rAPA* as a guide to packaging the plugin code.
- Update packaged customizations for rAPA to enable and configure any custom plugins added to an appliance. (For more information, see the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html].)
- Use the guidelines throughout this guide to help with troubleshooting plugin code throughout development and testing.

Chapter 3. Plugin Development Environment

Plugin development can be performed on the same system as other appliance development work, but it requires additional software and should be managed in separate directories from those used for developing appliances. Use the instructions in this chapter to set up the environment, including checking out the appropriate version of the rAPA source. Then, use instructions in the next chapter to start working with a template plugin.

When you complete the setup of the environment, your home directory on your development system should have contents as follows. Check off this list as you set up the environment:

- `~/conary/` -- Directory where appliance development contexts have been established
- `~/.bashrc` -- File used to configure the user's Bash shell in Linux systems, modified here to ensure the environment variable *PYTHONPATH* is set or appended to include the path to CherryPy
- `~/raa/` -- Directory created when checking out the rAPA source code from rPath
- `~/rapa-devel/` -- Directory for the custom development work for rAPA, including plugin development to be packaged separately for your appliance

Note

These instructions assume you have established an appliance development environment and understand the basic process of writing recipes and building packages for appliances based on the rPath Appliance Platform. They also assume you have added and customized the rPath Appliance Platform Agent for at least one appliance product. If this information is new to you, be sure to step through the *Application to Appliance: A Hands-on Guide* [http://wiki.rpath.com/wiki/Application_to_Appliance] and the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html], both available at docs.rpath.com [http://docs.rpath.com].

Supporting Software Setup

If you are developing plugins using the Application to Appliance Development Image, distributed from *app2app.rpath.org* [http://app2app.rpath.org] (a product on rBuilder Online), then you already have most of the software required for developing rAPA plugins or for the rare modification to rAPA code that requires rebuilding rAPA. If you are not using the app2app image, install the following in your Conary-based development environment:

- `group-rapa=raa.rpath.org@rpath:rapa-3` -- Having the package itself installed and functional on the system provides a quick reference to how rAPA is installed (such as the location of certain files) and the ability to develop the plugin directly on top of operational code. This is different than checking out the source used to build rAPA; the rAPA source has many files only used to build the code while the rAPA install represents the structure that accepts the separately-installed custom plugins.
- Dependencies brought in by `group-rapa` -- Because the rAPA code has some of the same requirements for building as it does for running, allow `group-rapa` to bring in all of its dependencies.

- `kid:runtime=conary.rpath.com@rpl:2` -- This component of the `kid` package is required for developing the web front-end of custom plugins, powered by the Kid templating system as used in rAPA (kid-templating.org [http://kid-templating.org/]).
- `sudo` -- This software is used by some of the scripts in the rAPA development environment. Be sure the current user can use the `sudo` command to perform tasks that require root access. This is already configured for *devuser* on the `app2app` development image; use an Internet search to find further documentation about the `sudo` utility and how to configure it so that other users can use it.
- `mercurial` -- Source code for rAPA is available for checkout from a version control system. rPath uses *Mercurial* (http://www.selenic.com/mercurial) as a version control system during code development, and then uses `r.addMercurialSnapshot` as part of the `rapa` Conary package recipe to check out the code from Mercurial before building the package from source. You will need `mercurial` installed to make a local checkout of that code so that your custom plugin code can successfully reference its Python objects.
- `make` and `xmlto` from `conary.rpath.org@rpl:2` -- These are required only in the case that you have to rebuild rAPA from its source; they are used to build some of the inline documentation in rAPA. However, rebuilding rAPA from source is not required to customize the contents displayed from the *Help* and *Learn More* links, as explained with instructions in the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html].

One additional piece required for plugin development is *CherryPy 3* (cherrypy.org [http://cherrypy.org/]), an HTTP framework for Python. This is installed as part of installing rAPA on the local system. (Note that prior versions of this guide included instructions for installing CherryPy from a tarball, though this should not be necessary when developing for any version of rAPA 3.x.) Use the following steps to set the *PYTHONPATH* variable in `~/.bashrc` to ensure that your development environment knows where CherryPy is installed on your system:

1. Before setting the variable, verify you have CherryPy contents installed as part of having installed rAPA on the local system:

   ```
   $> conary q rapa --ls | grep cherrypy
   ```

2. Use the path to the CherryPy directory, indicated in the previous query, to create or append the environment variable *PYTHONPATH* each time a new Bash shell is started (each time you log in). Open the file `~/.bashrc` and add the following line to the end of that file:

   ```
   export PYTHONPATH=/usr/lib/python2.4/site-packages/
   ```

3. Log in as root or use `sudo su -` to start working as root, and edit `/etc/sudoers` (which is set to read-only and can only be edited by root) to add the following line:

   ```
   Defaults env_keep="PYTHONPATH"
   ```

   This is necessary because some of the scripts in the rAPA development environment attempt to run as root and pass this environment variable through the `sudo` utility. (This step was

not required for versions of the `sudo` utility prior to 1.6.9, but recommended practice is to take this extra step anyway, which should work in all cases.)

In addition to the software described here, required for all plugin development, be sure to install any other software your custom plugin requires to perform its tasks. For example, if your plugin sets up a new database on the local system, be sure that the database software is installed and configured as it would be on the appliance at the time a rAPA user performs that task through rAPA.

rAPA Source Checkout Setup

All of the source code for rAPA is available in the `raa` Mercurial checkout from *hg.rpath.com*. However, to ensure you have the correct version of the source code associated with a given rAPA release, be sure to update the checkout with the tag associated with that release.

Note

Because of its previous branding, rAPA code still contains the string "raa" in many of its filenames, directory names, and object names. Both "rapa" and "raa" refer to the same product.

Use the following steps to check out `raa` and to update the checkout to reflect the source for rAPA 3.0.2.1:

1. Change to your home directory:

```
$> cd ~/
```

2. Clone (check out) the `raa` Mercurial repository from *hg.rpath.com*:

```
$> hg clone http://hg.rpath.com/raa
```

3. Change to the `raa` directory created by the clone operation:

```
$> cd raa
$> pwd
/home/devuser/raa
```

4. Show all the tags that identify version to which you can update the Mercurial checkout. Each of the versions listed on the left side of the screen represents a past or future release of rAPA:

```
$> hg tags
```

5. Update the repository to a tag from the list. For version 3.0.2.1, use the following command:

```
$> hg update 3.0.2.1
```

6. Run `make` in the `raa` directory to generate some file necessary to work out of this directory:

   ```
   $> make
   ```

   Be sure to run `make clean` before updating the checkout to a different version of rAPA, and to run `make` again after that update.

Over time, use other Mercurial (`hg`) commands to manage the checkout directory, including pulling the latest contents, listing the tags from the latest contents, and updating to a different tag to represent a different rAPA release. Use the `hg(1)` manual page for a complete listing of commands and options:

```
$> man hg
```

Note

rPath does not recommend developing plugins on the latest code in the `raa` checkout as it represents the tip of development for rAPA and may include code that is not part of a supported release. After a clone operation or pulling an update to the checkout, always update the checkout files to the Mercurial tag that represents a supported released version of rAPA.

Plugin Development Directory Setup

Set up a directory tree for plugin development work that is separate from appliance development work, and configure it to use rAPA code from your `raa` Mercurial checkout. This guide recommends using `~/rapa-devel` as the directory name and will use that in all examples. Use the following steps to set up this directory tree:

1. Create `~/rapa-devel` and change to that directory:

   ```
   $> mkdir ~/rapa-devel
   $> cd ~/rapa-devel
   ```

2. Use the following `ln` commands to create a series of soft links to your `raa` directory so that development work is separate and yet appropriately references the necessary code from rAPA:

   ```
   $> ln -s ../raa/dev.cfg
   $> ln -s ../raa/raa
   $> ln -s ../raa/raaplugins
   $> ln -s ../raa/raa-service
   ```

```
$> ln -s ../raa/start-raa-service.sh
$> ln -s ../raa/start-raa-web.sh
$> ln -s ../raa/raa_service_dev.conf
$> ln -s ../raa/raa/content content
$> ln -s ../raa/raa-web
```

3. Create a new file `~/rapa-devel/raa_service.conf` that has the following one line in it, replacing *username* with your Linux user name:

```
listenerUser          username
```

4. Create a new file `~/rapa-devel/custom.cfg` that has the following lines in it so that the rAPA service will run as the current user instead of its native "raa-web" user, replacing *username* with your Linux user name:

```
[global]
server.run_user = 'username'
server.run_group = 'username'
server.lock_path = '.'
```

Use the following as guidelines for working in this development directory:

- When working with existing code checked out in the appliance development environment, copy the necessary files from that context into this plugin development directory. If necessary, unpack the tar file containing the packaged code. For more detailed instructions, see the upcoming chapter Chapter 4, *Generate a Template Plugin*.
- When creating new directories in this tree, be sure to create `__init__.py` files (they can be empty) in each directory that contains files of Python code in that directory or in any of its subdirectories. This alerts Python to find and use the Python code under that directory. If you are starting with a template or existing plugin code, the `__init__.py` files should already exist.
- When packaging the plugin for installing on an appliance, copy the necessary files from the plugin development directory to the package development directory in the appliance development environment (under the appropriate Conary context). If desired, create a compressed tar file of the files to use in the Conary package. For more detailed instructions, see the upcoming chapter Chapter 10, *Package Custom Plugins for rAPA*.

Generate Test Certificates

Generate HTTPS certificates that are used when accessing rAPA over secure HTTP. A script in the rAPA checkout from rPath can be run to generate the certificates. Use the following commands to generate the certificates from the rAPA development directory:

```
$> cd ~/rapa-devel
$> ../raa/distro/bin/raa-gendummycert.sh $(pwd)/test.key $(pwd)
```

Plugin Generator Setup

The rPath Appliance Platform Agent includes a script that prompts for information and then generates a skeleton for your new plugin. In Chapter 4, *Generate a Template Plugin*, this guide instructs you to start your plugin with the plugin generator. Use the following commands o set up the plugin generator for use from your plugin development directory:

```
$> cd ~/rapa-devel
$> ln -s ../raa/plugingenerator.py && ln -s ../raa/plugintempla
```

Warning

The plugin generator code is not updated for every minor release of rAPA, so be sure to troubleshoot code as appropriate to ensure it works with the version of rAPA for which you are developing plugins.

Run rAPA in Development Mode

If you are using the *app2app* development image or a similar Conary-based system that has rAPA already installed, you must stop the `raa` service that is running on the system and then start the web and server services from the development directory instead.

First, as the root user, or using `sudo` or another means of having root permissions, stop the `raa` service:

```
$> sudo /sbin/service raa stop
Shutting down rPath Appliance Platform (Service Daemon):    [  C
Shutting down rPath Appliance Platform (Web Service):       [  C
```

Then, use the "start" scripts, linked from `~/rapa-devel`, to start each of the two script-driven services from the development environment. Each service will require a separate shell from that which you are using for other development work, each which should have the *PYTHONPATH* environment variable set after having completed the steps in the section called “Supporting Software Setup”). You need to be able to switch to these separate shells to stop and start the services when testing plugins during development. The following are two ways to operate in separate shells in a way that allows you to stop and start each service throughout the development process:

- Use two new secure shell (ssh) logins to the development system and run a service in each. OpenSSH is already installed for use in the `app2app` development image so that the system can be accessed by ssh.
- Use separate terminals running on the same system, and use the Alt key plus one of the first six function keys (F1 through F6) on your keyboard to switch between the terminals on the same system. This can be used readily in the `app2app` development image and other images based on rPath Linux and running in multi-user mode (or "runlevel 5," the default mode of most systems).

After deciding how to switch between terminals or shells, use the following steps to launch the two development-mode rAPA services from those additional shells:

1. In one shell, use the following commands to launch the web service:

```
$> cd ~/rapa-devel
$> ./start-raa-web.sh
```

2. After the web service is running successfully, use the the following commands in another shell to launch the server service:

```
$> cd ~/rapa-devel
$> ./start-raa-service.sh
```

3. Continue running the services in the foreground of each shell instead of putting them in the background to return the shell prompt. By keeping them in the foreground, they can monitored for status and error messages, and they can be stopped quickly when necessary by using Ctrl-C. While the services are running, the rAPA interfaces are accessible, such as the web interface which can be accessed in a web browser using the following URL (replacing "hostname" with the hostname or IP of the development system):

```
https://hostname:8003
```

These two additional shells running these services from the development environment must be launched each time you want to test your custom plugin code. rPath recommends testing your plugin code often throughout the process to see how it works in rAPA, minimizing the number of new or modified lines of code that you need to troubleshoot between tests. When the scripts are running, the rAPA web and XML-RPC interfaces should be running on the system just as they would be if the `raa` service from the installed rAPA was running.

Note

If any shell displays an error that it cannot locate CherryPy, be sure the PYTHONPATH environment variable is set as indicated in previous sections of this chapter. Use `echo $PYTHONPATH` at any time to see the value of the variable in the current shell.

To return to using rAPA as it is installed on the system instead of from the development environment, stop the two development scripts and restart the `raa` service on the system.

Chapter 4. Generate a Template Plugin

After establishing an environment as described in Chapter 3, *Plugin Development Environment*, generate a template for each plugin you need to develop. This chapter uses the plugin generator Python script as set up with the instructions in the section called "Plugin Generator Setup".

First, use the plugin generator by running the Python script linked in `~/rapa-devel`, and entering information as prompted by the script. The following is an example of running the script:

```
$> python plugingenerator.py
Vendor directory name (valid python): samplecorp
Plugin directory name (valid python): exampletask
Display name: Example Task
Tooltip: Example Task
Name of the class (valid python): ExampleTask
```

After completing the script prompts as shown in the previous example, the directory `~/rapa-devel/samplecorp` exists with the skeleton `exampletask` plugin code. The information prompted in the script fills in the new template as follows:

- *Vendor directory name* -- Developers traditionally group all their custom plugins into a single directory, often named to reference the corporation developing and releasing the appliance. Select a name that serves as a valid Linux directory name as well as a valid path which Python code can reference. If the directory already exists when the script is run, the new plugin will be placed in that existing directory.
- *Plugin directory name* -- Developers should name both the plugin directory and the Python class to complement the name of the task to be performed through rAPA. For the plugin directory, select a name that serves as a valid Linux directory name as well as a valid path which Python code can reference.
- *Display name* -- Type the task title as it should appear in the rAPA web interface. If this task is only made available through XML-RPC, do not skip this step, but provide the title of the task as it should appear in other interfaces.
- *Tooltip* -- Type the tooltip that should be displayed when a user hovers a mouse pointer over the name of the task in the rAPA web interface. If this task is only made available through XML-RPC, provide the same value as the display name from the previous step.
- *Name of the class* -- Developers should name both the plugin directory and the Python class to complement the name of the task to be performed through rAPA. For the Python class, select a "CamelCase" name that follows appropriate Python syntax for class names. For example, the rAPA standard plugin that provides the *Updates* task is in the `updatetroves` directory and the Python class name is `UpdateTroves`.

Change to the new plugin directory to begin code work, and use the remaining chapters of this guide as a reference to developing the web and server parts of the plugin, the plugin help, and the package used to install the plugin on your appliance. The following is a summary of the plugin skeleton generated by the plugin generator:

- `__init__.py` -- The presence of this file, typically an empty file, indicates to Python that Python code is in that directory or one or more of its subdirectories. Each new directory created

for your plugin should have one `__init__.py` file, though most plugins will only require the four directories laid out by the plugin generator.

- `help` -- This directory is empty, but it is the location in which you can create help for the task provided by the plugin, which is typically accessed from a "Learn More" link in the task's web interface. For further instructions on developing help for your plugin, see Chapter 9, *Add Task Help*.
- `templates` -- This directory has an `index.kid` file which is the Kid template that lays out the task's web interface and calls the objects in the web part of the plugin and in the rAPA code. For further instructions on laying out the web interface for your plugin, see Chapter 5, *Task Interface Layout*.
- `web` -- This directory has a Python file with a filename and class based on your responses to the generator, and this part of the plugin is for class objects that power the direct user interaction of the task. The web part of the plugin may be able to perform all the necessary plugin actions, or it can call one or more functions from the server part of the plugin (in sibling directory `srv`). For instructions on developing the web part of the plugin, see Chapter 6, *Function Development for the Web Part*, and for exposing web functions to the XML-RPC interface, see Chapter 8, *Expose Plugin Functions for the XML-RPC Interface*.
- `srv` -- This directory has a Python file with a filename and class based on your responses to the generator, and this part of the plugin is for class objects that power longer-running back-end functions or functions that require root permissions on the system. The server part of the plugin is designed to be called only by the objects from the web part of the plugin. For instructions on developing these server back-end objects, see Chapter 7, *Function Development for the Server Part*.

Chapter 5. Task Interface Layout

Before programming the functions of the rAPA task driven by your plugin, modify the Kid template that lays out the web interface for the task. This exists as `templates/index.kid` in the skeleton directory as created by the plugin generator. Be sure the rAPA user is prompted for all necessary information and that the physical layout works in all the web browsers you intend to support for administration. Use the sections that follow as a guide in programming this layout in the Kid template.

Note

If this task should only be available through the XML-RPC interface, you can skip this interface layout. However, be sure to use the information from the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html] for ensuring the task is enabled, but hidden from the web interface: http://docs.rpath.com/rAPA_Customization_Guide/sect-available_tasks.html.

JavaScript Handling

The Kid templating system will interpret the `index.kid` file as XML first before assembling the XHTML, JavaScript, and Python that will render in a web browser. As a result, certain elements of JavaScript will be handled differently than expected when developing XHTML directly.

The most common issue when putting JavaScript into the XML is that the less-than character ("<") is interpreted as the opening of an XML tag. This will cause tracebacks when attempting to render the web interface for your plugin in rAPA, even if the code is valid JavaScript. A quick solution to this is to use the escape character < instead of the less-than character when you need to use less-than, such as when evaluating a variable for a loop. The following shows how this might look in a `for` loop:

```
for (var count = 0; count &lt; selectedItems.options.length; co
    if (selectedItems.options[count].selected) {
        selectedArray.push(selectedItems.options[count].value);
    }
}
```

There are other options for escaping conflicting JavaScript code when editing the XML in `index.kid`, such as using "CDATA" sections or XML comment notation around the code. See documentation on XML syntax for more information about these and other escape options.

View the Template Layout

As instructed in setting up your plugin development environment, you should be running two services from scripts in the development directory. Also, as instructed in generating your template

plugin, you should have a directory for developing plugins out of that same rAPA development environment. To view the layout of your template, restart the web service script which is running in one of your alternate shells (`start-raa-web.sh`). Use Ctrl-C to stop the script, use the up-arrow key to reveal the previous command (`./start-raa-web.sh`), and press Enter to launch the script again.

After restarting the web service, refresh the rAPA web interface in your web browser (`https://hostname:8003` where "hostname" is the DNS hostname or IP address of your development system). Any plugins created in the development directory (using the plugin generator) or soft-linked from that directory (the `raaplugins` directory linked to the Mercurial checkout), should show up in the web interface in its default configuration.

As you develop the layout, be sure to restart the web service script each time you want to view your changes. Click the name of the task in the rAPA menu to access the layout.

To test configuration options all plugins, including any that you build in to your custom plugin, use the information from the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html] as a guide. Any modifications you would package in the `custom.cfg` file for rAPA can be test in the `custom.cfg` file you created in the development directory. This configuration only applies to this development-mode version of rAPA.

Set Up Task Headings and Text

The main heading for the task is automatically populated by the responses provided to the plugin generator. Other headings and text have placeholders with generic text. The plugin generator was not designed to make use of the all of the current CSS for rAPA, some of these placeholders can be removed completely and replaced by *div* items that make use of the classes in the rAPA CSS.

Replace the placeholder ("h3" and "h5") headers with *div* entries that imitate the code in the `.kid` files for the standard plugins (checked out at `~/raa/raaplugins/`. Look for the following in those `.kid` files to see how they are used to set up task headings and text:

- `<?python instructions = "" ?>` (optional) -- Place this somewhere within the "body" tags and fill in the quotes with the instructions that should be displayed at the top of the task layout. These instructions typically describe the task and instruct the user how to use it. If you are using this feature, be sure to include the *displayinstructions div* described in the *page-content* item to follow.
- `div class="plugin-page" id="plugin-page"` -- This is the *div* which should house the entire plugin body. The plugin generator does not create this *div*, so be sure to add its beginning and ending tags around your layout code.
- `div class="page-content"` -- This is the *div* that sets the page layout for the task as it appears in the web inteface. In addition, if you want to include the instructions set with the Python variable `instructions` as previously described, be sure to include the following *div* just after opening the *page-content div*:

```
<div py:replace="display_instructions(instructions, r
```

- `div class="page-section"` -- This *div* lays out the header for a subsection on the same page. The plugin interface using the default rAPA CSS typically has at least one of these, and you can add as many subsections as you need. Close this *div* immediately after the title of the subsection.

  ### Note

  As an alternative to subsections on the same page, you can use a tabbed interface for subtasks that need more display space, as described in the section called “Tabbed Interface Layout”.

- `div class="page-section-content"` -- This *div* lays out the content of a subsection on the page. This should include all the content of the subsection, including any instructional text for the subtask and any of the form elements required to get and process user input. For additional *div* tags and JavaScript elements that could be nested in the subsection, see the section called “Form Elements”.

The following example code replaces the header and form placeholders in `~/rapa-devel/samplecorp/exampletask/templates/index.kid`:

```
<?python
    instructions = "These are the instructions for my task."
?>

<div class="plugin-page" id="plugin-page">

    <div class="page-content">
        <div py:replace="display_instructions(instructions, raa

    <!-- Subsection to enable and schedule the example task -->
    <div class="page-section">
        Enable and Schedule the Example Task
    </div>

    <div class="page-section-contents">
        <div class="form-line">
            Enable example function:
            <input onclick="javascript: toggleDisable();" id="e
            <input onclick="javascript: toggleDisable();" id="e
        </div>
        <label id="scheduleLabel">Schedule the Example Function
        <div class="schedule">
            ${RepeatScheduleWidget(schedule, True)}
        </div>
    </div>

    <script type="text/javascript">
        addLoadEvent(function() {enableSched(schedEnabled)});
```

```
        </script>

        <a href="kick">Run the example function now</a>
        </div>

    </div>
```

Note

If you copy and paste from the program listing above, be sure to know what code you should already have in place in your `index.kid` file, and you should adjust the "input" and "py" code lines to ensure they do not have any carriage returns (lines should wrap naturally, but without a return character added to the end of a wrapped line). Always be sure the file has well-formed code in accordance with XHTML, JavaScript, and other standards.

The example code displayed above results in a web interface for the example task resembling the following image (from a default rAPA 3.0.2.1 web interface):

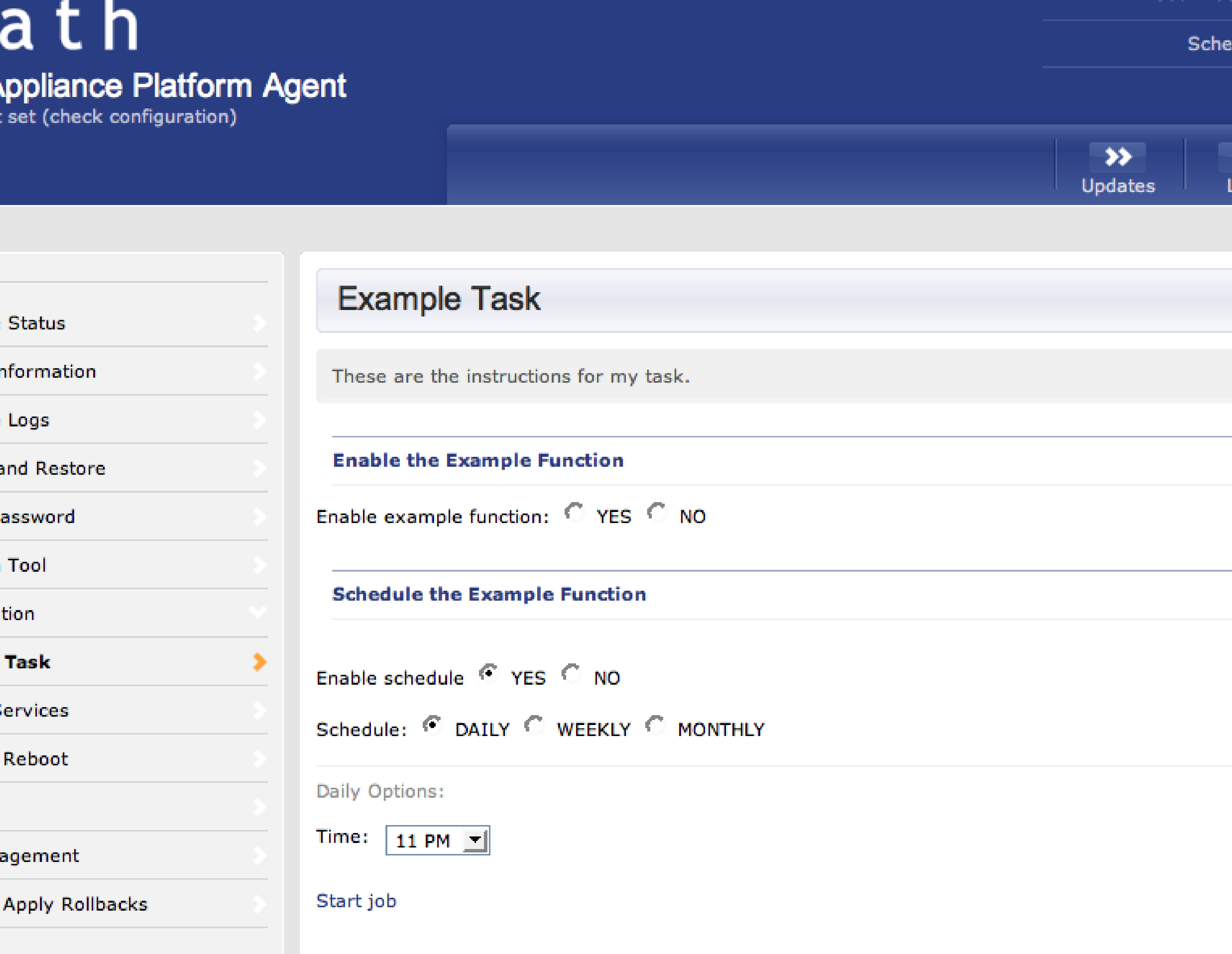

Figure 5.1. Example Plugin Initial Layout

Form Elements

This section gives examples of how to add some common form elements to the task interface while making use of the rAPA API.

Text Entry Field

The rAPA CSS styles some *div* classes to lay out some form elements, including the JavaScript inputs with for text entry. As a reference, note how the *edit-line-top* and *edit-line* `div` classes are used in the layout of the User Management task (`raaplugins/usermanagement/index.kid`). The following is an example of lines that prompt for text entry:

```
<!-- Subsection for text entry -->
<div class="page-section">
     Text Entry
</div>
<div class="page-section-contents">
    <div class="edit-line-top">
        <label for="lastname" class="last-name">Last name:<
        <input type="text" id="lastname" name="lastname" va
    </div>
    <div class="edit-line">
        <label for="firstname" class="first-name">First nam
        <input type="text" id="firstname" name="firstname"
    </div>
</div>
```

Figure 5.2. Example Text Entry

Text entry placeholders should be used at this step before working on the code that is used to process the entries (covered in Chapter 6, *Function Development for the Web Part*).

Drop-down List

The rAPA code can use a combination of JavaScript's `select` tags and Python code to populate selections in a drop-down list. Each plugin that includes such a list typically defines the items in that list with imported Python code from other parts of the plugin, or from a configuration value from a `custom.cfg` file. Any interface actions taken upon selection are defined with standard JavaScript functions in the `head` part of the `index.kid` file. As a reference, note how the `select` and tags are used in the layout of the Logs task (`raaplugins/logs/index.kid`).

The following is an example of lines that define a JavaScript function in the `head` and that set up a drop-down list in one of the task's subsections. This example shows JavaScript alone, but the Logs task and some of the information in later chapters of this guide show how Python is used to define the options for the list list displayed:

```
<head>
<script type="text/javascript">
    function dropdown()
    {
        var droplist = document.getElementById("exampleDrop
        document.getElementById("txtdisplay").value = dropl
    }
</script>
</head>
<!-- Other code truncated for easy viewing in this example

<!-- Subsection for drop-down list -->
<div class="page-section">
     Drop-down List
</div>
<div class="page-section-contents">
    Select one of the following:
    <select id="exampleDropDown" name="exampleDropDown" onc
        <option>One</option>
        <option>Two</option>
        <option>Three</option>
    </select>
    <p>You current selection is:
        <input type="text" id="txtdisplay" size="20" /></p>
</div>
```

wn List

of the following: Three

nt selection is: Three

Figure 5.3. Example Drop-down

Note

If desired, JavaScript can be defined in a separate `.js` file and included with `src` in the `script` tags instead of being programmed inside the `index.kid` file. However,

unless the scripts are extensive, keeping plugin-specific code together in the same file may make the plugin easier to manage over time. The following is an example of pulling in source from a separate JavaScript file assumed to be in the same directory with the index file installed on the appliance:

```
<script type="text/javascript" src="example.js"></sc
```

Highlight Selection List

A highlight selection list uses similar code to that of the drop-down list in that it uses the `select` tags for JavaScript code. For this multiple-selection option, though, the `select` tags have `multiple="multiple"` and `size` set equal to the height of the box displayed before it will scroll. As a reference, note how the `select` and tags are used for the "Roles" selection in the user group management part of the User Management task (`raaplugins/usermanagement/groups.kid`).

The HTML portion of the following is similar to the drop-down list example except with more options and the ability to shift-click or control-click (or command-click on Macs) to select multiple items in the list. The JavaScript function in the `head` is used to assemble the values in the list for display on clicking the *Show My Selections* list. As with the drop-down example, this example shows JavaScript alone, but the User Management task and some of the information in later chapters of this guide show how Python is used to define the options for the list displayed:

```
<head>
<script type="text/javascript">
function listSelected()
{
    var selectedValuesObj = document.getElementById('select
    var selectedArray = new Array();
    var selectedItems = document.getElementById('exampleMul
    for (var n = 0; n &lt; selectedItems.options.length; n+
        if (selectedItems.options[n].selected) {
            selectedArray.push(selectedItems.options[n].val
        }
    }
    selectedValuesObj.value = selectedArray;
}
</script>
</head>
<!-- Other code truncated for easy viewing in this example

<!-- Subsection for highlighted-selection list -->
<div class="page-section">
```

```
    Highlighted Selection List
</div>

<div class="page-section-contents">
    Select one or more of the following:
    <select id="exampleMultiSelect" name="exampleMultiSelec
        <option>One</option>
        <option>Two</option>
        <option>Three</option>
        <option>Four</option>
        <option>Five</option>
    </select>
    <p><input type="button" value="Show My Selections" oncl
    <p>Selections: 
        <input type="text" id="selectedValues" size="40" />
</div>
```

Figure 5.4. Example Highlighted Selection

Scheduler Tool and Radio Buttons

A radio button selection assumes that if one item in a grouping is selected, all others will be unselected. A rAPA plugin can use the JavaScript `input` code with `type="radio"` to set up a single radio button in the selection, and can tie the radio buttons together with with corresponding `name` values. As a reference, note how the `input` and tags are used for the "Backup Settings" selection in the Back Up and Restore task (`raaplugins/backup/templates/config.kid`).

The plugin generator includes scheduler code with one such radio button selection which prompts for a schedule entry if *Enable* is selected. The following example shows this same code with some modifications to better work with the code in rAPA 3.0.2.1:

```
<body>

<!-- Function defined in the body for execution on page loa
<script type="text/javascript">
    var schedEnabled = ${enable};
    function enableSched(enabled)
    {
        if(!enabled) {
            addElementClass('scheduleLabel', 'disabled');
            hideElement('schedule');
        }
        else{
            removeElementClass('scheduleLabel', 'disabled')
            showElement('schedule');
        }
    }
</script>

<div class="page-section">
    Enable and Schedule the Example Task
</div>

<div class="page-section-contents">
    <div class="form-line">
        Enable the example tasks: 
        <input class='radio' onclick="javascript:enableSche
        <input class='radio' onclick="javascript:enableSche
    </div>

    <!-- Subsection to schedule the task when enabled -->
    <label id="scheduleLabel">Schedule this Task</label>
    <div id="schedule">
        ${RepeatScheduleWidget(schedule, True)}
    </div>
</div>
<script type="text/javascript">
    addLoadEvent(function() {enableSched(schedEnabled)});
</script>

</body>
```

The following images show the section in both "enabled" and "disabled" states:

nd Schedule the Example Task

example tasks: Enabled Disabled

edule YES NO

DAILY WEEKLY MONTHLY

ns:

PM

Figure 5.5. Example Schedule Enabled

nd Schedule the Example Task

example tasks: Enabled Disabled

Figure 5.6. Example Schedule Disabled

Checkbox Selection

Unlike radio buttons, code for checkboxes assumes that any number of items may be selected at one time. A rAPA plugin can use the JavaScript `input` code with `type="checkbox"` to set up a single checkbox in the selection. As a reference, note how the `input` and tags are used for the "Start on Boot" column for each service in the Manage Services task (`raaplugins/services/index.kid`).

The following code shows how to use a checkbox selection to make similar choices to those in the example for the highlighted selection list example:

```
<body>
<!-- Subsection for checkbox selection -->
<div class="page-section">
    Checkbox Selection
</div>
```

```
<div class="page-section-contents">
    Select one or more of the following:
    <table>
        <tr>
            <td>
                <input type="checkbox" class="check" value=
            </td>
            <td>
                One
            </td>
        </tr>
        <tr>
            <td>
                <input type="checkbox" class="check" value=
            </td>
            <td>
                Two
            </td>
        </tr>
        <tr>
            <td>
                <input type="checkbox" class="check" value=
            </td>
            <td>
                Three
            </td>
        </tr>
    </table>
</div>
</body>
```

x Selection

or more of the following:

Figure 5.7. Example Checkbox Selection

As with all of these form elements, the functions that drive the behavior could be calls to other parts of the plugin code. Be sure to reference the combination of Python and JavaScript that is

used in the standard plugins as well as in the upcoming chapter Chapter 6, *Function Development for the Web Part*.

Date and Time Selection

Repeating tasks that require a regular schedule use the rAPA scheduler, and plugins can prompt for managing that schedule with the code described in the section called “Scheduler Tool and Radio Buttons”. However, for tasks that require a one-time scheduled task using the scheduler, or that just need a quick way to select a date and time for any use, can use the rAPA API tools for selecting a date and time. As a reference, note JavaScript and imported objects from `raa` are used for scheduling a reboot or shutdown using the Schedule Reboot task (`raaplugins/reboot/reboot.kid`).

Status and Dialog Boxes

Some tasks require additional feedback from the user to verify a selection or to report the status of a task in progress. Python code can be written and called from the `index.kid` file to display status in the web interface or to display a dialog box that disables the web interface while prompting for a user response. As a reference, see the imported objects from the rAPA API and the plugin code in the Rollbacks task (`raaplugins/rollbacks/index.kid`).

As a quick and consistent way to add status feedback to a plugin, rAPA standard plugins import `CallbackDisplayWidget` from `raa.templates.callbackdisplaywidget` and use it to automatically format and display messages associated with the current task. This code is already written as a Kid template in the core rAPA code and can be referenced at `~/raa/raa/templates/callbackdisplaywidget.kid`.

Tabbed Interface Layout

There are some extra steps required to create a tabbed interface for the web interface of a custom rAPA plugin. Use the following steps to set up multiple tabs in the interface:

1. Copy `index.kid` to create other Kid template files in the `templates` directory, each with a name in appropriate Python syntax.
2. For each of the new Kid template files, be sure to create a new definition in the `web` part of the plugin. In most cases, it is sufficient to just copy the *index* method definition created by the plugin generator and changing each instance of "index" in the copy to the name of the new Kid file (minus the `.kid` extension). For example, when adding `advanced.kid` to the `exampletask` plugin, the file `samplecorp/exampletask/web/exampletask.py` is edited as follows. :

```
@raa.web.expose(allow_xmlrpc=True, template="samplecorp
def index(self):
    return self._getConfig()

@raa.web.expose(allow_xmlrpc=True, template="samplecorp
def advanced(self):
```

```
return self._getConfig()
```

3. In the same directory with the Kid template files, create a Python file for controlling the labels used on the tabs. By placing this code in a separate file, it can be maintained in a single file instead of separately in each Kid template file. This guide uses the filename `tabcontrol.py` for the file (`samplecorp/exampletask/templates/tabcontrol.py`).
4. In the tab control file, assign values to a variable used to label the tabs in the interface. The variable is (in Python terms) a dictionary of tuples and is imported to each Kid file which, in turn, passes the value to the rAPA code that lays out the tabbed interface. This guide uses the `tabLabels` variable. The following is example contents of `tabcontrol.py`:

```
tabLabels = {
    'index': "Index",
    'advanced': "Advanced Options"
}
```

5. In all the Kid template files, be sure to import two items in the *head* part of the file alongside any other widgets that have been imported: the tabbed page "widget" from rAPA, and the variable value that is holding the labels for the tabs. For the example used in this guide, the following shows shows the lines that must be included in *head* in each file (at minimum, though other Python lines can be included, too):

```
<?python
    from raa.templates.tabbedpagewidget import Tabb
    from tabcontrol import tabLabels
?>
```

6. Within the *page-content div* in each Kid template file, call `TabbedPageWidget` as appropriate for that file's role in the interface. The following shows the call in `index.kid` for the `exampletask` plugin:

```
${TabbedPageWidget(forcepage='index', value=tabLabe
```

The following shows the call in `advanced.kid` for that same `exampletask` plugin:

```
${TabbedPageWidget(forcepage='advanced', value=tabLa
```

rPath recommends laying out this tabbed interface alongside other layout programming before moving on to programming the functions in the interface.

Chapter 6. Function Development for the Web Part

Though there is current documentation available at the rPath Wiki, this location will soon include more extensive information specific to this version of rAPA. In the meantime, use the following to link to the rPath Wiki plugin development documentation: http://wiki.rpath.com/wiki/rPath_Appliance_Platform_Agent:Plugin_D evelopment [http://wiki.rpath.com/wiki/rPath_Appliance_Platform_Agent:Plugin_Development]

Chapter 7. Function Development for the Server Part

Though there is current documentation available at the rPath Wiki, this location will soon include more extensive information specific to this version of rAPA. In the meantime, use the following to link to the rPath Wiki plugin development documentation: http://wiki.rpath.com/wiki/rPath_Appliance_Platform_Agent:Plugin_Development

Chapter 8. Expose Plugin Functions for the XML-RPC Interface

To a task operation to the rAPA web interface, identify the methods used to perform the task as programmed in the web part of the plugin, and use the following Python decorator just before the definition. This assumes you have imported `raa.web` in the same Python file:

```
@raa.web.expose(allow_xmlrpc=True)
```

This is the minimum requirement to expose the method through XML-RPC. (In addition, `allow_json=True` is added so that the method can be called with JSON.) When a method is "exposed," it can be called as described in the section of the *rAPA Customization Guide* [http://docs.rpath.com/rAPA_Customization_Guide/index.html] that describes using rAPA over XML-RPC and setting up an XML-RPC client. As long as your plugin is included and enabled in rAPA on your appliance, even if the web interface is hidden, these exposed methods can be called over XML-RPC. On the other hand, if a method is not "exposed," it is not available through the rAPA XML-RPC interface.

The following is an example of the Status plugin code that shows system information. Note the decorators used, including `raa.web.expose` with `allow_xmlrpc=True`:

```
@raa.web.expose(allow_xmlrpc=True, template="raaplugins.status.temp
@raa.web.require(raa.authorization.NotAnonymous())
def sysinfo(self):
    ret = dict()
    ret['du'] = self._du()
    ret['sysinfo'] = self._systemStatus()
    return ret
```

Chapter 9. Add Task Help

MORE TO COME... this is new content that does not currently exist on the rPath Wiki.

Chapter 10. Package Custom Plugins for rAPA

This chapter assumes you are already familiar with basic packaging of software for Conary filesystem management, as introduced in *Application to Appliance: A Hands-on Guide* [http://wiki.rpath.com/wiki/Application_to_Appliance]. It also assumes you have an appliance development environment configured for packaging software for an appliance. With those assumptions, use the following instructions to package a plugin to be installed alongside rAPA and other software for an appliance:

1. Run `make clean` and `make` on the rAPA code one last time to generate the plugin files to package.
2. If you want to close your plugin source code, you may want to take two additional steps here before packaging the plugin from the files in the development environment:
   a. Save the source code to your own version control system or other separate directory tree to preserve it so that you can use the code to make changes in the future. This source code should include all Python source files (".py"), the files based on the Kid templating system (".kid"), and the XML source used for the plugin help (".xml").
   b. From the plugin development environment, remove the ".xml" file used for the plugin help, and remove any ".py" files that were only used to build the ".pyc" files (everything except the "init" files). Be sure to leave the ".kid" files in place.
3. Create a compressed tar file of the files in the plugin directory for the plugin you want to package. You can package multiple plugins in the same package, though creating a single package for each plugin ensures that they can be installed, tested, and maintained individually. For example, if the plugin files have been developed in the directory `~/rapa-devel/samplecorp/exampletask/`, use the following command to create the compressed tar file (tarball):

```
$> cd ~/rapa-devel/samplecorp
$> tar zcvf exampletask.tar.gz exampletask
```

4. In the context for creating new packages for your appliance, create a new package and change to that package directory. For example, the following commands change to the appliance development environment for the `example` product and create a package called `example-rapa-plugins`:

```
$> cd ~/conary/example/example-1-devel/
$> rmake newpkg example-rapa-plugin
```

5. Copy the new plugin tarball to the new package directory. The previous examples can apply the following command:

```
$> cp ~/rapa-devel/samplecorp/exampletask.tar.gz ~/cona
```

6. Change to the new package directory and write a recipe to package and install the plugin on an appliance that also includes rAPA:

```
$> cd ~/conary/example/example-1-devel/example-rapa-plu
$> vi example-rapa-plugin.recipe
```

Use the following example recipe as a guide. Notice that the custom plugin is installed to a vendor-specific directory under `/usr/lib/raa/`. Be sure that all your vendor-specific custom plugins are installed to the same location for easier administration:

```
# Example Plugin
class ExampleRapaPlugin(PackageRecipe):
    name = 'example-rapa-plugin'
    version = '1.0'

    def setup(r):
        r.addArchive('exampletask.tar.gz',
                     dir='/usr/lib/raa/samplecorp/')
```

If your plugin requires additional software to be installed on the appliance in order to perform the task in rAPA, add `r.Requires` policy actions to the recipe to create dependencies that Conary will resolve automatically when the package is installed (and when the appliance is built). See the Conary API documentation for more about this action: http://cvs.rpath.com/conary-docs/conary.build.packagepolicy.Requires-class.html.

7. Build the package, then create changesets to test the package install on a test system that has the appropriate version of rAPA installed. For the example, the following commands are used:

```
$> rmake build example-rapa-plugin
$> rmake changeset 1 example-rapa-plugin.ccs
```

8. After successful testing, commit the package to the appliance product repository, then use an `r.add` line in the appliance group recipe to add the new package to the appliance. Rebuild the appliance and test as appropriate.