# OpenShift Enterprise 2 Cartridge Specification Guide

Specifications for developing OpenShift Enterprise cartridges

Red Hat OpenShift Documentation Team

# OpenShift Enterprise 2 Cartridge Specification Guide

## Specifications for developing OpenShift Enterprise cartridges

Red Hat OpenShift Documentation Team

## Legal Notice



## Abstract

The OpenShift Enterprise Cartridge Specification Guide provides guidelines and specifications about components and services that are essential to the design of custom cartridges. This guide helps developers to create custom cartridges so that application components that are not currently supported by the available cartridges can be integrated with OpenShift Enterprise.

# Table of Contents

# Chapter 1. Introduction to OpenShift Enterprise

OpenShift Enterprise by Red Hat is a Platform as a Service (PaaS) that provides developers and IT organizations with an auto-scaling, cloud application platform for deploying new applications on secure, scalable resources with minimal configuration and management overhead. OpenShift Enterprise supports a wide selection of programming languages and frameworks, such as Java, Ruby, and PHP. Integrated developer tools, such as Eclipse integration, JBoss Developer Studio, and Jenkins, support the application life cycle.

Built on Red Hat Enterprise Linux, OpenShift Enterprise provides a secure and scalable multi-tenant operating system for today's enterprise-class applications while providing integrated application runtimes and libraries.

OpenShift Enterprise brings the OpenShift PaaS platform to customer data centers, enabling organizations to implement a private PaaS that meets security, privacy, compliance, and governance requirements.

# Chapter 2. Managed Files

Managed files are files that have non-default settings, or that require special handling by OpenShift Enterprise.

The **`$cartridge_name/metadata/managed_files.yml`** file lists managed files and strings that OpenShift Enterprise uses during different stages of the cartridge life cycle.

## File Patterns

Most entries in the **`managed_files.yml`** file use file patterns. OpenShift Enterprise treats these patterns like shell globs. **`Dir.glob`** processes entries that contain asterisk (*) symbols using the **`File::FNM_DOTMATCH`** flag. It treats entries that end in a forward slash (/) as directories, and other entries as files. For more information on globs, see http://ruby-doc.org/core-1.9.3/Dir.html#method-c-glob.

Entries that begin with **`~/`** start at the gear directory. All other entries start at the cartridge directory.

## Strings

Some entries allow string values. These values return directly without any modification.

## Allowed Entries

OpenShift Enterprise supports the following entries:

**Table 2.1. Managed File Entries**

| Entry | Type | Usage | Reference |
|---|---|---|---|
| locked_files | File Pattern | Files that the application developer can read but not update. | Chapter 3, *Locking Cartridges* |
| snapshot_exclusions | File Pattern | Array of file names not to backup from the gear when **`rhc snapshot`** runs. | Chapter 10, *Backing Up and Restoring Cartridges* |
| setup_rewritten | File Pattern | Files that OpenShift Enterprise removes before running **`setup`**. | Section 6.2, "setup Script" |
| process_templates | File Pattern | ERB templates OpenShift Enterprise renders after running **`setup`**. | Section 6.1, "Embedded Ruby (ERB) Processing" |
| restore_transforms | Strings | Set of regex transforms for rewriting file names when **`rhc restore`** runs. | Chapter 10, *Backing Up and Restoring Cartridges* |

The following example demonstrates the structure of a basic **`managed_files.yml`** file:

**Example 2.1. managed_files.yml**

```
locked_files:
- env/
- ~/.foorc
snapshot_exclusions:
- mydir/*
```

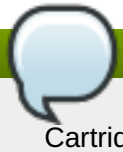
```
restore_transforms:
- s|${OPENSHIFT_GEAR_NAME}/data|app-root/data|
process_templates:
- **/*.erb
setup_rewritten:
- conf/*
```

# Chapter 3. Locking Cartridges

Cartridge instances on a gear are either *locked* or *unlocked* at any given time. Locking a cartridge enables cartridge scripts to have greater access to the gear's files and directories. Application developers have read and write access to unlocked files, and read-only access to locked files. This means that application scripts and hooks cannot override cartridge code when the cartridge is locked.

OpenShift Enterprise controls the lock state of cartridges, moving them between locked and unlocked at various points in the cartridge life cycle.

A cartridge with no **`locked_files`** entry in the **`$cartridge_name/metadata/managed_files.yml`** file is permanently unlocked. This approach is not recommended, but it may be sufficient for simple cartridges.



**Note**

Cartridge file locking is not a security measure. It is designed to prevent application developers from accidentally breaking their applications by modifying cartridge files.

## 3.1. Cartridge Lock Configuration

The **`locked_files`** entry in the **`$cartridge_name/metadata/managed_files.yml`** file lists files and directories that OpenShift Enterprise locks at certain points during the cartridge life cycle.

If a file in the **`locked_files`** list does not exist, OpenShift Enterprise creates the file before your **`setup`** script is called. OpenShift Enterprise also creates missing directories if required.

If files require application developers to have read and write access to them while an application is deploying and running, do not allow OpenShift Enterprise to create them from the **`locked_files`** list. For example, create **`~/.node-gyp`** and **`~/.npm`** in a node.js cartridge using a **`setup`** or **`install`** script.

Entries that begin with **`~/`** start at the gear directory. All other entries start at the cartridge directory. Entries that end with a forward slash (/) are treated as directories. Entries that end with an asterisk (*) are treated as lists of files. Entries that end with any other character are treated as files.

**Note**

OpenShift Enterprise does not change entry types. For example, if you enter a directory without a forward slash (/) at the end, OpenShift Enterprise treats it as a file. A cartridge can fail to operate if its **`locked_files`** entries are not accurate.

**Example 3.1. PHP locked_files Configuration Entry**

```
locked_files:
- ~/.pearrc
- bin/
- conf/*
```

Explanation:

- **`~/.pearrc`**: when locked, you can edit this file but application developers cannot.
- **`php/bin/`**: the directory is locked but not the files it contains. Only you can add files to the directory, but both you and application developers can edit those files.
- **`php/conf/*`**: the directory is not locked, but the files in the directory are locked. Both you and application developers can add files to the directory, but only you can edit them.

## Reserved Files

All visible files and directories in a gear's home directory are reserved. Certain hidden files are also reserved. While a cartridge is unlocked, you can create any unreserved hidden file or directory in the gear's home directory.

### Reserved Hidden Files

- **`~/.ssh`**
- **`~/.sandbox`**
- **`~/.tmp`**
- **`~/.env`**

# Chapter 4. Exposing Services

Most cartridges provide services by binding to ports. Cartridges must declare to which ports they bind, and provide variable names to describe:

- The IP addresses provided to the cartridge for binding.
- The local gear ports to which the cartridge services bind.
- (Optional) The public proxy ports that expose local gear ports for communication between related gears in an application, such as the TCP proxy public endpoint.
- (Optional) TCP endpoint mappings that establish a front end for application users.

## 4.1. TCP Endpoints

TCP endpoints are services that are exposed by a cartridge, and are accessible by other cartridges or gears in an application. They may be any protocol which uses TCP, such as **`http`** or **`mysql`**. These services may also be exposed to application users through mappings. OpenShift Enterprise only creates endpoint ports for scalable applications.

The endpoints can be defined in the **`Endpoints`** section of the **`$cartridge_name/metadata/manifest.yml`** file.

**Example 4.1. Endpoints Entry**

```
Endpoints:
  - Private-IP-Name:   <name of IP variable>
    Private-Port-Name: <name of port variable>
    Private-Port:      <port number>
    Public-Port-Name:  <name of public port variable>
    Protocols:         [<protocol type 1>,<protocol type 2>]
    Mappings:
      - Frontend:      '<frontend path>'
        Backend:       '<backend path>'
        Options:       { ... }
      - <...>
  - <...>
```

When a cartridge is installed on a gear, it automatically assigns IP addresses to each IP variable name ensuring that assigned addresses can bind to the specified port.

If an endpoint specifies a public port variable, a public port proxy mapping is created using a random external port accessible through the gear's DNS entry.

### Endpoint Environment Variables

Endpoint values are exposed to cartridge scripts and application code through environment variables. These environment variables are formed from the **`Cartridge-Short-Name`** element and from the endpoint variable names specified in the **`manifest.yml`** file.

**Example 4.2. Environment Variable Format**

```
OPENSHIFT_{Cartridge-Short-Name}_{name of IP variable}          =>
<assigned internal IP>
OPENSHIFT_{Cartridge-Short-Name}_{name of port variable}        =>
<endpoint specified port>
OPENSHIFT_{Cartridge-Short-Name}_{name of public port variable} =>
<assigned external port>
```

### Endpoint Protocols

You can define the protocols for services using the **`Protocols`** variable. **`Protocols`** takes a comma-separated list of protocol types from the following available options:

**Table 4.1. Endpoint Protocols**

| Protocol | Description |
|---|---|
| tcp | TCP |
| http | HTTP |
| https | HTTP Secure (HTTP over SSL/TLS) |
| ws | WebSocket |
| wss | WebSocket Secure (WebSocket over SSL/TLS) |
| tls | SNI Proxy |
| mongodb | MongoDB |
| mysql | MySQL |
| postgresql | PostgreSQL |

If the **`Protocols`** list is not set, the default behavior matches the pre-**`Protocols`** behavior. For example, if an endpoint has **`Mappings`**, assume HTTP; otherwise, assume TCP. The front-end modules also translate **`Mappings`** options. For example, if a **`Mappings`** entry has **`websocket`** set in its **`Options`**, then **`ws`** is added to the **`Protocols`** list.

### Endpoint Mappings

If an endpoint specifies **`Mappings`**, a front-end **`httpd`** route to the cartridge is created for each mapping entry using the provided options. The **`Frontend`** key is a front-end path element connected to a back-end URI specified by the **`Backend`** key. The **`Options`** hash enables additional route configuration options.

**Table 4.2. Endpoint Mapping Options**

| Option | Description |
|---|---|
| websocket | Enable WebSocket on a particular path |
| gone | Mark the path as gone (URI is ignored) |
| forbidden | Mark the path as forbidden (URI is ignored) |
| noproxy | Mark the path as not proxied (URI is ignored) |
| redirect | Use redirection to URI instead of proxy (URI must be a path) |
| file | Ignore request and load file path contained in URI (must be a path) |
| tohttps | Redirect request to HTTPS and use the path contained in the URI (must be a path) |

## 4.2. TCP Endpoint Example

This section provides an example **Endpoints** entry in a **$cartridge_name/metadata/manifest.yml** file, and demonstrates how OpenShift Enterprise uses this entry to create environment variables, public proxy port mappings, and **httpd** routes.

**Example 4.3. Endpoints Entry**

```
Name: CustomCart
Cartridge-Short-Name: CUSTOMCART

...

Endpoints:
  - Private-IP-Name:   HTTP_IP
    Private-Port-Name: WEB_PORT
    Private-Port:      8080
    Public-Port-Name:  WEB_PROXY_PORT
    Protocols:         [ws]
    Mappings:
      - Frontend:      '/web_front'
        Backend:       '/web_back'
      - Frontend:      '/socket_front'
        Backend:       '/socket_back'
        Options:       { "websocket": true }

  - Private-IP-Name:   HTTP_IP
    Private-Port-Name: ADMIN_PORT
    Private-Port:      9000
    Public-Port-Name:  ADMIN_PROXY_PORT
    Protocols:         [http]
    Mappings:
      - Frontend:      '/admin_front'
      - Backend:       '/admin_back'

  - Private-IP-Name:   INTERNAL_SERVICE_IP
    Private-Port-Name: 5544
    Public-Port-Name:  INTERNAL_SERVICE_PORT
```

### Environment Variables

Several environment variables are created for the cartridge using the information in the **Endpoints** entry.

**Example 4.4. Environment Variables**

```
# Internal IP/port allocations
OPENSHIFT_CUSTOMCART_HTTP_IP=<assigned internal IP 1>
OPENSHIFT_CUSTOMCART_WEB_PORT=8080
OPENSHIFT_CUSTOMCART_ADMIN_PORT=9000
OPENSHIFT_CUSTOMCART_INTERNAL_SERVICE_IP=<assigned internal IP 2>
OPENSHIFT_CUSTOMCART_INTERNAL_SERVICE_PORT=5544
```

```
# Public proxy port mappings
OPENSHIFT_CUSTOMCART_WEB_PROXY_PORT=<assigned public port 1>
OPENSHIFT_CUSTOMCART_ADMIN_PROXY_PORT=<assigned public port 2>
```

### Proxy Port Mapping

Proxy port mapping is assigned using the information in the **`Endpoints`** entry.

**Example 4.5. Proxy Port Mapping**

```
<assigned external IP>:<assigned public port 1> =>
OPENSHIFT_CUSTOMCART_HTTP_IP:OPENSHIFT_CUSTOMCART_WEB_PORT
<assigned external IP>:<assigned public port 2> =>
OPENSHIFT_CUSTOMCART_HTTP_IP:OPENSHIFT_CUSTOMCART_ADMIN_PORT
```

### httpd Routing

The **`httpd`** routes are assigned using the **`Endpoints`** entry.

**Example 4.6. httpd Routing**

```
http://<app dns>/web_front    =>
http://OPENSHIFT_CUSTOMCART_HTTP_IP:8080/web_back
http://<app dns>/socket_front =>
http://OPENSHIFT_CUSTOMCART_HTTP_IP:8080/socket_back
http://<app dns>/admin_front  =>
http://OPENSHIFT_CUSTOMCART_HTTP_IP:9000/admin_back
```

## 4.3. Custom HTTP Services

With ERB templates you can expose cartridge services using an application's URL by placing the Apache configuration code in the **`httpd.d`** directory.

After OpenShift Enterprise runs the cartridge **`setup`** script, it processes each ERB template and writes the contents of the node's **`httpd`** configuration.

**Example 4.7. A mongodb.conf.erb File**

```
Alias /health <%= ENV['OPENSHIFT_HOMEDIR'] +
"/mongodb/httpd.d/health.html" %>
Alias / <%= ENV['OPENSHIFT_HOMEDIR'] + "/mongodb/httpd.d/index.html" %>
```

## 4.4. Enabling Custom Paths for Websockets

Websockets are used to create real-time events initiated by an OpenShift Enterprise application.

The ability to add a custom path for websocket URLs to a cartridge must be enabled in that cartridge's **manifest.yml** file before it can be used in a new application. Add the following information to the **/usr/libexec/openshift/cartridges/*Cart_Name*/metadata/manifest.yml** file of the desired cartridge:

```
- Private-IP-Name: IP2
  Private-Port-Name: PORT2
  Private-Port: 8080
  Public-Port-Name: PROXY_PORT2
  Protocols:
  - http
  - ws
  Mappings:
  - Frontend: '/file_path'
    Backend: '/file_path2'
    Options:
      websocket: true
```

This adds a second endpoint to a cartridge with **ws** listed in the protocols and websockets set to **true**.

After a cartridge has been modified to use custom paths for websocket URLs, a new application can then be created with the modified cartridge. The application is accessible using the new endpoint, as shown in the following example:

```
ws://app-domain.example.com:8000/file_path
```

# Chapter 5. Creating Template Directories for Language Cartridges

Use the **$cartridge_name/template/** or **$cartridge_name/template.git/** directory to provide a basic example of an application written in the language or framework your cartridge packages. Welcome the application developer to your cartridge and inform them that the cartridge is operating correctly.

If you provide a **$cartridge_name/template.git/** directory, OpenShift Enterprise copies the directory for the application developer.

If you provide a **$cartridge_name/template/** directory, OpenShift Enterprise uses it to create a Git repository for the application developer. Ensure that your **setup** and **install** scripts account for the path change from **template** to **template.git**.

Create **.gitignore** files in empty directories to ensure the directories are retained when OpenShift Enterprise builds the Git repository.

**Example 5.1. Ruby Template Directory**

A Ruby 1.8 cartridge with Passenger support has a **template/public/** directory and a **config.ru** file to define the web application.

```
+- template
|  +- config.ru
|  +- public
|  |  +- .gitignore
|  .openshift
|  +- markers
|  |- ...
```

## 5.1. Marker Files

The **$cartridge_name/template/.openshift/markers/** directory contains example marker files for application developers. These files set conditions for various stages of a cartridge's life cycle. You can add marker files as required to enable application developers to control aspects of your cartridge.

**Example 5.2. Ruby 1.8 Marker Files**

| Marker | Action |
|---|---|
| force_clean_build | OpenShift Enterprise removes previous output from the **bundle install --deployment** command and reinstalls all gems according to the current **Gemfile** and **Gemfile.lock** files. |
| hot_deploy | OpenShift Enterprise serves new application code without restarting the application's web server. |
| disable_auto_scaling | OpenShift Enterprise does not automatically scale a scalable application. |

## 5.2. Action Hooks

The **$cartridge_name/template/.openshift/action_hooks/** directory contains examples of application developer action hooks that run during the cartridge life cycle.

**Example 5.3. Action Hooks**

```
pre_start_`cartridge name`
post_start_`cartridge name`
pre_stop_`cartridge name`
```

OpenShift Enterprise runs default **action_hooks** as indicated in the **control** script. To add additional hooks, run them explicitly in the **control** script. Ensure appropriate documentation is available for application developers to use the additional hooks correctly.

If you find that action hooks are not working, they may not be executable. To fix this, run the following command:

```
$ git update-index --chmod=+x .openshift/action_hooks/file_name
```

The **--chmod=(+/-)x** command sets the execute permissions on the updated file specified.

# Chapter 6. Using Cartridge Scripts

Cartridge scripts act as the application programming interface (API) for a cartridge. Use these scripts to contain the required code for single version software that configures easily. For complex configurations and software with multiple versions, use these scripts as shim code to set up the required environment and run additional scripts. You can also create symbolic links from these scripts.

Cartridge scripts are located in the **`$cartridge_name/bin/`** directory, and run from the cartridge's home directory.

**Table 6.1. Required Scripts**

| Script Name | Usage |
|---|---|
| setup | Creates and configures files that OpenShift Enterprise copies from the cartridge repository to the gear's directory. Runs for the initial installation and every upgrade. |
| control | Enables OpenShift Enterprise or the application developer to control the state of a cartridge and its packaged software. |

**Table 6.2. Optional Scripts**

| Script Name | Usage |
|---|---|
| teardown | Prepares the gear for cartridge removal. |
| install | Creates and configures files that OpenShift Enterprise copies from the cartridge repository to the gear's directory. Runs only on the first installation of the cartridge. |
| post-install | Configures the cartridge after the cartridge starts. Runs only on the first installation of the cartridge. |
| metrics | Gathers cartridge level metrics. |

## 6.1. Embedded Ruby (ERB) Processing

Embedded Ruby (ERB) is a templating system that embeds Ruby into a text document. To provide more flexible configuration and environment variable options, OpenShift Enterprise enables you to provide certain values as ERB templates. For more information on ERB templates, see http://ruby-doc.org/stdlib-1.9.3/libdoc/erb/rdoc/ERB.html.

OpenShift Enterprise renders ERB templates at **`safe_level 2`** and processes them in two passes. For more information on Ruby safe levels, see http://www.ruby-doc.org/docs/ProgrammingRuby/html/taint.html.

### ERB Processing Passes

1. The first pass processes entries in the **`$cartridge_name/env/`** directory. This pass is mandatory, and occurs before OpenShift Enterprise runs the **`$cartridge_name/bin/setup`** script.

2. The second pass processes entries specified in the **`process_templates`** entry of the **`$cartridge_name/metadata/managed_files.yml`** file. This pass occurs after OpenShift Enterprise runs the **`$cartridge_name/bin/setup`** script, but before it runs the **`$cartridge_name/bin/install`** script. This enables the **`setup`** script to create or modify ERB templates as required, and for the **`install`** script to use the processed values.

> **Example 6.1. Environment Variable Template**

For OpenShift Enterprise release 2.0, an **env/OPENSHIFT_MONGODB_DB_LOG_DIR.erb** file contains:

```
<%= ENV['OPENSHIFT_MONGODB_DIR'] %>/log/
```

For OpenShift Enterprise release 2.1 and later, an **env/OPENSHIFT_MONGODB_DB_LOG_DIR.erb** file contains:

```
<%= ENV['OPENSHIFT_LOG_DIR'] %>
```

The value of **LOG_DIR** for each cartridge is set to the same value as **OPENSHIFT_LOG_DIR**.

From that ERB file, OpenShift Enterprise creates an **env/OPENSHIFT_MONGODB_DB_LOG_DIR** environment variable containing:

```
/var/lib/openshift/aa9e0f66e6451791f86904eef0939e/mongodb/log/
```

**Example 6.2. php.ini Configuration Template**

A **conf/php.ini.erb** file contains:

```
upload_tmp_dir = "<%= "#{ENV['OPENSHIFT_PHP_DIR']}tmp/" %>"
session.save_path = "<%= "#{ENV['OPENSHIFT_PHP_DIR']}sessions/" %>"
```

From that ERB file, OpenShift Enterprise creates a **conf/php.ini** file containing:

```
upload_tmp_dir =
"/var/lib/openshift/aa9e0f66e6451791f86904eef0939e/php/tmp/"
session.save_path =
"/var/lib/openshift/aa9e0f66e6451791f86904eef0939e/php/sessions/"
```

Other possible uses for ERB templates are **includes** values in **httpd** configuration files, database configuration values for storing persistent data in the **OPENSHIFT_DATA_DIR** directory, and the application name value in the **pom.xml** file.

## 6.2. setup Script

### Synopsis

**setup** [ --version <version> ]

### Options

**--version <version>**: selects which version of the cartridge to install. If you do not supply a version, OpenShift Enterprise installs the version given in the **Version** element of the **$cartridge_name/metadata/manifest.yml** file.

### Description

The **setup** script creates and configures files that OpenShift Enterprise copies from the cartridge repository to the gear's directory. The **setup** script must be re-entrant. OpenShift Enterprise runs the script for every upgrade that is not backward compatible. Add logic you want to run only once to the **install** script, not the

**setup** script.

Add files created during setup to the **setup_rewritten** section of the **$cartridge_name/metadata/managed_files.yml** file. During an upgrade, OpenShift Enterprise deletes these files prior to running the **setup** script.

If you use ERB templates to configure software, OpenShift Enterprise processes these files for environment variable substitution after running the **setup** script.

*Lock context: unlocked.*

## 6.3. install Script

### Synopsis

**install** [ --version <version> ]

### Options

**--version <version>**: selects which version of the cartridge to install. If you do not supply a version, OpenShift Enterprise installs the version given in the **Version** element of the **$cartridge_name/metadata/manifest.yml** file.

### Description

The **install** script creates and configures files that OpenShift Enterprise copies from the cartridge repository to the gear's directory. OpenShift Enterprise runs the **install** script only on the first installation of a cartridge.

Put logic for one-time operations, for example generating passwords, creating ssh keys, and adding environment variables, in the **install** script.

Report client results and messages with the **install** script, not the **setup** script.

The **install** script may substitute a version dependent of the **template** or **template.git** directories.

*Lock context: unlocked.*

## 6.4. post-install Script

### Synopsis

**post-install** [ --version <version> ]

### Options

**--version <version>**: selects which version of the cartridge to install. If you do not supply a version, OpenShift Enterprise installs the version given in the **Version** element of the **$cartridge_name/metadata/manifest.yml** file.

### Description

Use the **post-install** script to configure your cartridge after the cartridge starts. OpenShift Enterprise only runs the **post-install** script for the first installation of the cartridge.

*Lock context: locked.*

## 6.5. teardown Script

### Synopsis

**`teardown`**

### Description

The **`teardown`** script prepares the gear for cartridge removal. The script only runs when OpenShift Enterprise removes the cartridge from a gear; it does not run when OpenShift Enterprise deletes the gear. The gear continues to operate without the functionality of the removed cartridge.

*Lock context: unlocked.*

## 6.6. control Script

### Synopsis

**`control <action>`**

### Options

**`<action>`**: the action the cartridge performs.

### Description

The **`control`** script enables OpenShift Enterprise or the application developer to control the state of a cartridge and its packaged software.

**Table 6.3. Control Script Actions**

| Action | Result |
|---|---|
| update-configuration, pre-build, build, deploy, post-deploy | See Chapter 9, *OpenShift Build Process*. |
| start | Starts the software the cartridge controls. |
| stop | Stops the software the cartridge controls. |
| status | Returns a zero (0) exit status if the cartridge code is running. |
| reload | Instructs the cartridge and its packaged software to reload their configuration information. This action only operates if the cartridge is running. |
| restart | Stops the current process and starts a new one for the packaged software. |
| threaddump | Signals the packaged software to perform a thread dump, if applicable. |
| tidy | Releases unused resources. |
| pre-snapshot | Prepares the cartridge for a snapshot. |
| post-snapshot | Tidies the cartridge after a snapshot. |
| pre-restore | Prepares the cartridge for restoration. |
| post-restore | Tidies the cartridge after restoration. |

*Lock context: locked.*

### Using the `tidy` Action

By default, the **`tidy`** action performs the following operations:

- Garbage collects the Git repository.
- Removes all files in the **`/tmp`** directory.

Add additional operations to the **`tidy`** action by editing the **`tidy()`** function in the **`$cartridge_name/bin/control`** file. Because applications have limited resources, it is recommended that you tidy thoroughly.

**Example 6.3. Additional `tidy` Operations**

```
rm $OPENSHIFT_{Cartridge-Short_Name}_DIR/logs/log.[0-9]
```

```
cd $OPENSHIFT_REPO_DIR ; mvn clean
```

### Using the `status` Action

When an application developer queries the status of your packaged software, use a zero (0) exit status to indicate correct operation. Direct information to the application developer using **`stdout`**. Return errors with a non-zero exit status using **`stderr`**.

OpenShift Enterprise maintains the expected state of an application in the **`~/app-root/runtime/.state`** file. Do not use this file to determine the status of the packaged software as it contains the expected state of the application, not the current state.

**Table 6.4. Values for `.state`**

| Value | Status |
|---|---|
| building | Application is building. |
| deploying | Application is deploying. |
| idle | Application is shutdown due to inactivity. |
| new | A gear exists, but no application is installed. |
| started | Application started. |
| stopped | Application is stopped. |

## 6.7. metrics Script

With the release of OpenShift Enterprise 2.1, a metrics entry can be added to the cartridge's **`$cartridge_name/metadata/manifest.yml`** to inform OpenShift Enterprise that it supports metrics.

**Example 6.4. Metrics Entry**

```
Metrics:
- enabled
```

The **`metrics`** script must be an executable file in the **`$cartridge_name/bin/`** directory.

### Message Format

A metrics message must include the following fields and be written to standard out (STDOUT):

```
type=metric <metric name>=<metric value>
```

**Example 6.5. Metrics Message Example**

```
type=metric thread.count=5
```

## 6.8. Exit Status Codes

OpenShift Enterprise follows the convention that scripts return zero (0) for success and non-zero for failure.

OpenShift Enterprise supports special handling of several non-zero exit codes. These codes enable OpenShift Enterprise to refine its behavior, for example when returning **HTTP** status codes through the REST API or when deciding whether to continue or abort an operation.

If a cartridge script returns a value not included in the following tables, OpenShift Enterprise treats the error as fatal to the cartridge.

**Table 6.5. User Errors**

| Exit Code | Usage |
|---|---|
| 1 | Non-specific error |
| 97 | Invalid user credentials |
| 99 | User does not exist |
| 100 | An application with specified name already exists |
| 101 | An application with specified name does not exist and cannot be operated on |
| 102 | A user with login already exists |
| 103 | Given namespace is already in use |
| 104 | User's gear limit has been reached |
| 105 | Invalid application name |
| 106 | Invalid namespace |
| 107 | Invalid user login |
| 108 | Invalid SSH key |
| 109 | Invalid cartridge types |
| 110 | Invalid application type specified |
| 111 | Invalid action |
| 112 | Invalid API |
| 113 | Invalid auth key |
| 114 | Invalid auth iv |
| 115 | Too many cartridges of one type per user |
| 116 | Invalid SSH key type |
| 117 | Invalid SSH key name or tag |
| 118 | SSH key name does not exist |
| 119 | SSH key or key name not specified |
| 120 | SSH key name already exists |
| 121 | SSH key already exists |

| Exit Code | Usage |
|---|---|
| 122 | Last SSH key for user |
| 123 | No SSH key for user |
| 124 | Could not delete default or primary key |
| 125 | Invalid template |
| 126 | Invalid event |
| 127 | A domain with specified namespace does not exist and cannot be operated on |
| 128 | Could not delete domain because domain has valid applications |
| 129 | The application is not configured with this cartridge |
| 130 | Invalid parameters to estimates controller |
| 131 | Error during estimation |
| 132 | Insufficient Access Rights |
| 133 | Could not delete user |
| 134 | Invalid gear profile |
| 135 | Cartridge not found in the application |
| 136 | Cartridge already embedded in the application |
| 137 | Cartridge cannot be added or removed from the application |
| 138 | User deletion not permitted for normal or non-subaccount user |
| 139 | Could not delete user because user has valid domain or applications |
| 140 | Alias already in use |
| 141 | Unable to find nameservers for domain |
| 150 | A plan with specified id does not exist |
| 151 | Billing account was not found for user |
| 152 | Billing account status not active |
| 153 | User has more consumed gears than the new plan allows |
| 154 | User has gears that the new plan does not allow |
| 155 | Error getting account information from billing provider |
| 156 | Updating user plan on billing provider failed |
| 157 | Plan change not allowed for subaccount user |
| 158 | Domain already exists for user |
| 159 | User has additional filesystem storage that the new plan does not allow |
| 160 | User max gear limit capability does not match with current plan |
| 161 | User gear sizes capability does not match with current plan |
| 162 | User max untracked additional filesystem storage per gear capability does not match with current plan |
| 163 | Gear group does not exist |
| 164 | User is not allowed to change storage quota |
| 165 | Invalid storage quota value provided |
| 166 | Storage value not within allowed range |
| 167 | Invalid value for **`nolinks`** parameter |
| 168 | Invalid scaling factor provided. Value out of range. |
| 169 | Could not completely distribute **`scales_from`** to all groups |
| 170 | Could not resolve DNS |
| 171 | Could not obtain lock |
| 172 | Invalid or missing private key is required for SSL certificate |
| 173 | Alias does exist for this application |
| 174 | Invalid SSL certificate |
| 175 | User is not authorized to add private certificates |
| 176 | User has private certificates that the new plan does not allow |

| Exit Code | Usage |
|---|---|
| 180 | This command is not available in this application |
| 181 | User maximum tracked additional filesystem storage per gear capability does not match with current plan |
| 182 | User does not have **`gear_sizes`** capability provided by current plan |
| 183 | User does not have **`max_untracked_addtl_storage_per_gear`** capability provided by current plan |
| 184 | User does not have **`max_tracked_addtl_storage_per_gear`** capability provided by current plan |
| 185 | Cartridge **`X`** can not be added without cartridge **`Y`** |
| 186 | Invalid environment variables: expected array of hashes. |
| 187 | Invalid environment variable **`X`**. Valid keys **`name`** (required), **`value`** |
| 188 | Invalid environment variable name **`X`**: specified multiple times |
| 189 | Environment name **`X`** not found in application |
| 190 | Value not specified for environment variable **`X`** |
| 191 | Specify parameters **`name/value`** or **`environment_variables`** |
| 192 | Environment name **`X`** already exists in application |
| 193 | Environment variable deletion not allowed for this operation |
| 194 | Name can only contain letters, digits and underscore and cannot begin with a digit |
| 210 | Cannot override existing location for Git repository |
| 211 | Parent directory for Git repository does not exist |
| 212 | Could not find **`libra_id_rsa`** |
| 213 | Could not read from SSH configuration file |
| 214 | Could not write to SSH configuration file |
| 215 | Host could not be created or found |
| 216 | Error in Git pull |
| 217 | Destroy aborted |
| 218 | Not found response from request |
| 219 | Unable to communicate with server |
| 220 | Plan change is not allowed for this account |
| 221 | Plan change is not allowed at this time for this account. Wait a few minutes and try again. If problem persists contact Red Hat support. |
| 253 | Could not open configuration file |
| 255 | Usage error |

**Table 6.6. Uncommon Server Errors**

| Exit Code | Usage |
|---|---|
| 140 | No nodes available. If the problem persists contact Red Hat support. |
| 141 | Cartridge exception. |
| 142 | Application is registered to an invalid node. If the problem persists contact Red Hat support. |
| 143 | Node execution failure. If the problem persists contact Red Hat support. |
| 144 | Error communicating with user validation system. If the problem persists contact Red Hat support. |
| 145 | Error communicating with DNS system. If the problem persists contact Red Hat support. |
| 146 | Gear creation exception. |

## 6.9. Communication Between OpenShift and Cartridges

A cartridge can provide services for use by multiple gears in one application. OpenShift Enterprise enables you to publish these services. Each message writes to **`stdout`** or **`stderr`** with an exit status, one message per line.

**Example 6.6. Service Messages**

```
ENV_VAR_ADD: <variable name>=<value>
```

```
CART_DATA: <variable name>=<value>
```

```
CART_PROPERTIES: <key>=<value>
```

```
APP_INFO: <value>
```

# Chapter 7. Environment Variables

OpenShift Enterprise uses environment variables to communicate information between cartridges, applications, and the system.

OpenShift Enterprise provides several system environment variables that are available for use at all cartridge entry points.

Place cartridge environment variables in the **`$cartridge_name/env/`** directory. OpenShift Enterprise loads cartridge variables after system environment variables, but before calling your code.

## 7.1. System Environment Variables

OpenShift Enterprise provides several system environment variables. These variables are read-only.

**Table 7.1. System Environment Variables**

| Name | Value |
|---|---|
| HOME | Alias for **`OPENSHIFT_HOMEDIR`**. |
| HISTFILE | Bash history file. |
| OPENSHIFT_APP_DNS | The fully qualified domain name of the application using your cartridge. |
| OPENSHIFT_APP_NAME | The name of the application using your cartridge. Assigned by the application developer. |
| OPENSHIFT_APP_UUID | The UUID of the application using your cartridge. Assigned by OpenShift Enterprise. |
| OPENSHIFT_DATA_DIR | The directory where the application and your cartridge store data. |
| OPENSHIFT_GEAR_DNS | The fully qualified domain name of the gear where your cartridge is installed. This may not be the same as **`OPENSHIFT_APP_DNS`**). |
| OPENSHIFT_GEAR_NAME | The name of the gear where your cartridge is installed. Assigned by OpenShift Enterprise. This may not be the same as **`OPENSHIFT_APP_NAME`**). |
| OPENSHIFT_GEAR_UUID | The UUID of the gear where your cartridge is installed. Assigned by OpenShift Enterprise. |
| OPENSHIFT_HOMEDIR | The home directory of the gear where your cartridge is installed. Assigned by OpenShift Enterprise. |
| OPENSHIFT_REPO_DIR | The directory where the application repository is stored. OpenShift Enterprise runs the application from this location. |
| OPENSHIFT_TMP_DIR | The directory where the application and your cartridge store temporary data. |
| TMP | Alias for **`OPENSHIFT_TMP_DIR`**. |
| TMPDIR | Alias for **`OPENSHIFT_TMP_DIR`**. |

## 7.2. Cartridge Environment Variables

OpenShift Enterprise provides three environment variables for all cartridges by default. These variables are read-only.

**Table 7.2. Cartridge Environment Variables**

| Name | Value |
|---|---|
| OPENSHIFT_{Cartridge-Short-Name}_DIR | The directory where cartridge information is installed. |
| OPENSHIFT_{Cartridge-Short-Name}_IDENT | The identity of the cartridge, sourced from its **`manifest.yml`** file. The format is **`Cartridge-Vendor:Version:Cartridge-Version`**. |
| OPENSHIFT_PRIMARY_CARTRIDGE_DIR | The directory where the primary cartridge on a gear is installed. For example, a scaling PHP application has both a PHP cartridge and a HAProxy cartridge installed on the head gear. In this case, the PHP cartridge is the primary cartridge. |

## 7.3. Custom Cartridge Environment Variables

You can add custom environment variables to a cartridge by adding them to the cartridge's **`$cartridge_name/env/`** directory or creating them with the cartridge's **`setup`** and **`install`** scripts.

Entries in a cartridge's **`$cartridge_name/env/`** directory do not override system-provided environment variables. Using system-provided environment variable names in the **`$cartridge_name/env/`** directory prevents the cartridge from installing correctly.

Prefix custom environment variables with **`OPENSHIFT_{cartridge short name}_`** to prevent overwriting other cartridge variables in the packaged software's process environment space.

Suffix directory environment variables with **`_DIR`** and the value with a backslash (/).

You can provide Embedded Ruby (ERB) templates for environment variables in the **`$cartridge_name/env/`** directory. OpenShift Enterprise processes ERB templates in this directory before calling the cartridge's **`setup`** script.

OpenShift Enterprise sets the **`PATH`** variable using the path **`/etc/openshift/env/PATH`**. If you provide an **`OPENSHIFT_{Cartridge-Short-Name}_PATH_ELEMENT`**, OpenShift Enterprise uses the value to build the **`PATH`** when your scripts run or an application developer performs an interactive log on.

**Important**

OpenShift Enterprise does not validate cartridge-provided environment variables. A cartridge can fail to function if its environment variable files contain invalid data.

### Packaged Software Environment Variables

If your cartridge packages software with its own environment variables, add these variables to the cartridge's **`$cartridge_name/env/`** directory or include them in the shim code of the scripts in the **`$cartridge_name/bin/`** directory.

**Example 7.1. Jenkins Environment Variables**

- **`JENKINS_URL`**
- **`JENKINS_USERNAME`**
- **`JENKINS_PASSWORD`**

# Chapter 8. Cartridge Events

OpenShift Enterprise provides a publish and subscribe system that enables a cartridge to act when a developer adds or removes another cartridge in an application.

The **Publishes** and **Subscribes** elements in the **$cartridge_name/metadata/manifest.yml** file detail support for cartridge events.

## 8.1. Cartridge Event Publishing

When OpenShift Enterprise adds a cartridge to an application, it uses entries in the **Publishes** section of the **$cartridge_name/metadata/manifest.yml** file to construct events sent to other cartridges in the application. Define publish events in the **manifest.yml** file using the following format:

```
Publishes:
  <event_name>:
    Type: "<event type>"
```

**Example 8.1. PHP Cartridge Publishes Entry**

```
Publishes:
  get-php-ini:
    Type: "FILESYSTEM:php-ini"
  publish-http-url:
    Type: "NET_TCP:httpd-proxy-info"
  publish-gear-endpoint:
    Type: "NET_TCP:gear-endpoint-info"
```

For each **Publishes** entry, OpenShift Enterprise runs a script named **$cartridge_name/hooks/$event_name**.

OpenShift Enterprise joins lines of output that the **hooks/$event_name** script writes to **stdout** with single spaces, then inputs the result to subscriber scripts in other cartridges that match the **Type** of the publish event. The input to matching subscriber scripts is prefaced with **hooks/<event_name> <gear_name> <namespace> <gear_uuid>**.

## 8.2. Cartridge Event Subscriptions

When OpenShift Enterprise adds a cartridge to an application, it uses entries in the **Subscribes** section of the **$cartridge_name/metadata/manifest.yml** file in other cartridges to determine what actions to take for those other cartridges. Define subscribe events in the **manifest.yml** file using the following format:

```
Subscribes:
  <event_name>:
    Type: "<event type>"
```

**Example 8.2. PHP Cartridge Subscribes Entry**

```
Subscribes:
  set-env:
    Type: "ENV:*"
    Required: false
  set-mysql-connection-info:
    Type: "NET_TCP:db:mysql"
    Required: false
  set-postgres-connection-info:
    Type: "NET_TCP:db:postgres"
    Required: false
  set-doc-url:
    Type: "STRING:urlpath"
    Required: false
```

When OpenShift Enterprise processes a cartridge publish script, it inputs the result to subscriber scripts in other cartridges that match the **Type** of the publish event. The input to matching subscriber scripts is prefaced with **$cartridge_name/hooks/<event_name> <gear_name> <namespace> <gear_uuid>**.

For each matching **Subscribes** entry, OpenShift Enterprise runs a script named **$cartridge_name/hooks/$event_name**. OpenShift Enterprise must send and process entries marked with **Required: true**.

The publisher script determines the format of the information input to the subscriber script. Ensure that subscriber script can parse the input correctly.

## 8.3. Cartridge Event Example

In this example, an application developer adds a MySQL database cartridge to a PHP application. The publish and subscribe relationship between the cartridges enables the PHP cartridge to set environment variables on its gear so it can connect to the new MySQL cartridge, which is on a different gear.

### MySQL Cartridge as Publisher

The MySQL cartridge lists a **publish-mysql-connection-info** event in the **Publishes** section of its **mysql/metadata/manifest.yml** file:

```
Publishes:
  publish-mysql-connection-info:
    Type: "NET_TCP:db:mysql"
```

The MySQL cartridge implements a script in **mysql/hooks/publish-mysql-connection-info**.

### PHP Cartridge as Subscriber

The PHP cartridge lists a **set-mysql-connection-info** event in the **Subscribes** section of its **php/metadata/manifest.yml** file:

```
Subscribes:
  set-mysql-connection-info:
    Type: "NET_TCP:db:mysql"
```

The PHP cartridge implements a script in **php/hooks/set-mysql-connection-info**.

### Cartridge Event Communication Process

OpenShift Enterprise matches the event **`Type`** in the PHP cartridge's **`Subscribes`** list to the event **`Type`** in the MySQL cartridge's **`Publishes`** list. In this example, the event **`Type`** is **`"NET_TCP:db:mysql"`**.

The MySQL cartridge's **`publish-mysql-connection-info`** script outputs the username, host, port, URL, and password required to connect to the MySQL instance:

```
OPENSHIFT_MYSQL_DB_USERNAME=username;
OPENSHIFT_MYSQL_DB_PASSWORD=password;
OPENSHIFT_MYSQL_DB_HOST=hostname;
OPENSHIFT_MYSQL_DB_PORT=port;
OPENSHIFT_MYSQL_DB_URL=url;
```

OpenShift Enterprise sends the output of the MySQL cartridge's **`publish-mysql-connection-info`** to the PHP cartridge's **`set-mysql-connection-info`** script using the following format:

```
hooks/publish-mysql-connection-info gear_name namespace gear_uuid
'OPENSHIFT_MYSQL_DB_USERNAME=username;OPENSHIFT_MYSQL_DB_PASSWORD=password;O
PENSHIFT_MYSQL_DB_HOST=hostname;OPENSHIFT_MYSQL_DB_PORT=port;OPENSHIFT_MYSQL
_DB_URL=url;'
```

Note that the publisher script determines the format of the information input to the subscriber script. When writing subscriber scripts, ensure that they parse the input correctly.

# Chapter 9. OpenShift Build Process

When an application developer pushes changes to an application's Git repository, OpenShift Enterprise builds and deploys the application using the updated repository. The build and deploy process changes if the application is scaling or if it uses a builder cartridge.

## 9.1. Default Build Life Cycle

If no builder cartridge is present, OpenShift Enterprise executes the default build life cycle when an application developer pushes changes to an application Git repository. The default life cycle consists of a **`build`**, **`preparation`**, **`distribute`**, and **`deploy`** phase.

In the default build life cycle, OpenShift Enterprise manages the starting and stopping of the application, and moves the updated code into **`$OPENSHIFT_REPO_DIR`**. The primary cartridge and application developer action hooks (**`$OPENSHIFT_REPO_DIR/.openshift/action_hooks`**) determine specific behaviors during this process.

**Build Phase**

During this phase, OpenShift Enterprise:

1. Runs the **`gear stop`** command to stop the application.
2. Runs the **`control pre-receive`** command on the primary cartridge.
3. Runs the **`control pre-repo-archive`** command on the primary cartridge.
4. Creates a new application directory: **`$OPENSHIFT_HOMEDIR/app-deployments/$date_$time`** and dependent subdirectories.

**Note**

If your cartridge requires a particular directory structure for dependencies, create a symbolic link for your cartridge directory structure into **`$OPENSHIFT_DEPENDENCIES_DIR`**. Use **`$OPENSHIFT_BUILD_DEPENDENCIES_DIR`** for build time only dependencies.

5. Copies **`$OPENSHIFT_HOMEDIR/app-root/runtime/dependencies`** from the active application to **`$OPENSHIFT_HOMEDIR/app-deployments/$date_$time/dependencies`**.
6. Removes previous applications starting from the oldest until the number set at **`$OPENSHIFT_KEEP_DEPLOYMENTS`** is reached.
7. Copies the new application source code to **`$OPENSHIFT_REPO_DIR`**. This is the only point in the build life cycle when OpenShift Enterprise copies the application source code.
8. Runs the **`control pre-build`** command on the primary cartridge.
9. Runs the **`pre-build`** user action hook, if present.
10. Runs the **`control build`** command on the primary cartridge.
11. Runs the **`build`** user action hook, if present.

**Preparation Phase**

1. OpenShift Enterprise runs the **`prepare`** user action hook, if present.
2. The application ID and checksum of application contents are calculated.
3. OpenShift Enterprise creates **`$OPENSHIFT_HOMEDIR/app-deployments/by-id/$deployment_id`** and points to **`../app-deployments/$date_time`**

**Distribute Phase**

1. OpenShift Enterprise synchronizes the new application with all child gears if the application is scalable.

**Deploy Phase**

During this phase, OpenShift Enterprise:

1. Updates **`$OPENSHIFT_HOMEDIR/app-root/runtime/repo`** so it points to **`../../app-deployments/$date_$time/repo`**
2. Updates **`$OPENSHIFT_HOMEDIR/app-root/runtime/dependencies`** so it points to **`../../app-deployments/$date_$time/dependencies`**
3. Runs the **`control update-configuration`** command on the primary cartridge.
4. Starts all secondary cartridges in the application.
5. Runs the **`control deploy`** command on the primary cartridge.
6. Runs the **`deploy`** user action hook, if present.
7. Starts the primary cartridge using the **`gear start`** command.
8. Runs the **`control post-deploy`** command on the primary cartridge.
9. Runs the **`post-deploy`** user action hook, if present.

**Result:**

The build is now complete and the application is running.

## 9.2. Default Scaling Build Life Cycle

On the head gear, where the web proxy runs, the build phase for a scalable application is the same as the default build phase for a non-scaling application. The deploy phase for scalable applications is different.

**Deploy Phase**

1. OpenShift Enterprise starts the secondary cartridges on the application's head gear.
2. OpenShift Enterprise runs the web proxy's **`deploy`** hook on the head gear.
3. The web proxy runs deployment steps on the application's secondary gears. For example, the default web proxy, HAProxy, preforms the following steps:
    a. It stops the secondary gears.
    b. It synchronizes the code and build artifacts from the head gear to the secondary gears.

c. It runs the primary cartridge's **`control update-configuration`** command on the secondary gears.

d. It starts all the secondary cartridges on the secondary gears.

e. It runs the primary cartridge's **`control deploy`** command on the secondary gears.

f. It runs the **`deploy`** user action hook, if present, on the secondary gears.

g. It starts the primary cartridge on the secondary gears. The application is now running on the secondary gears.

h. It runs the primary cartridge's **`control post-deploy`** command on the secondary gears.

i. It runs the **`post-deploy`** user action hook, if present, on the secondary gears.

4. OpenShift Enterprise runs the primary cartridge's **`control deploy`** command on the head gear.
5. OpenShift Enterprise runs the **`deploy`** user action hook, if present, on the head gear.
6. OpenShift Enterprise starts the primary cartridge on the head gear.
7. OpenShift Enterprise runs the primary cartridge's **`control post-deploy`** command on the head gear.
8. OpenShift Enterprise runs the **`post-deploy`** user action hook, if present, on the head gear.

The build is now complete, and the scaled application is running.

## 9.3. Builder Cartridge Life Cycle

If an application includes a builder cartridge, OpenShift Enterprise does not perform build tasks. Instead, the builder cartridge runs the build process.

During the Git **`pre-receive`** hook, OpenShift Enterprise runs the builder cartridge's **`control pre-receive`** command.

During the Git **`post-receive`** hook, OpenShift Enterprise runs the builder cartridge's **`control post-receive`** command.

**Note**

Build processes use the application developer's gear resources to run. When implementing a builder cartridge, do not copy source code or build artifacts more than necessary.

## 9.4. Archiving Applications

Current applications can be archived and re-deployed later. Use the **`$rhc archive-deployment`** command to archive applications:

```
$ rhc archive-deployment
```

## 9.5. Binary Deployment

Binary deployment is very similar to build and deploy without the build. Instead, the built artifacts and dependencies are provided and the deploy steps start at **`prepare`**. Binary deployment are enabled using:

```
$ rhc app configure $app --deployment-type binary
```

# Chapter 10. Backing Up and Restoring Cartridges

OpenShift Enterprise provides **snapshot** and **restore** features for user applications. These features enable OpenShift Enterprise application developers to:

- Snapshot the current state of an application to create a backup.
- Restore an application from an archived state.
- Copy or rename an application by taking a snapshot, creating a new application, then restoring the snapshot data to the new application.

## 10.1. Snapshot

When an application developer runs the **rhc snapshot save** command, OpenShift Enterprise creates an archive of the application and performs the following steps:

1. Stops the application by running the **gear stop** command.
2. Runs the **control pre-snapshot** command for each cartridge on the gear. You can control cartridge serialization in the snapshot by implementing the **control pre-snapshot** command in conjunction with exclusions. For example, you can snapshot to a database dump instead of a database file.
3. Builds a list of exclusions from the **snapshot_exclusions** entry in the **$cartridge_name/metadata/managed_files.yml** file for each cartridge on the gear.
4. Creates an archive of the application in **tar.gz** format and writes it to **stdout** for use by the client tools. In addition to the files listed in the **snapshot_exclusions** entry in the **managed_files.yml** file, OpenShift Enterprise excludes the following files:
   - Selected gear user files: **.tmp**, **.ssh**, **.sandbox**.
   - Application state file: **app-root/runtime/.state**.
   - Bash history file: **$OPENSHIFT_DATA_DIR/.bash_history**.
5. Runs the **control post-snapshot** command for each cartridge on the gear. Use this script to cleanup after the snapshot runs.
6. Will either stop or start the gear based on the state of the application before the snapshot.

### Snapshot Exclusions

Use the optional **snapshot_exclusions** entry in the **$cartridge_name/metadata/managed_files.yml** file to list files to exclude from the snapshot and restore process. File patterns originate from the **OPENSHIFT_HOMEDIR** directory, not the cartridge directory. Do not exclude files that your cartridge requires to operate.

**Example 10.1. snapshot_exclusions Entry**

```
snapshot_exclusions:
- mydir/*
```

OpenShift Enterprise uses the **tar** command when performing snapshots. See the **tar** man page **--exclude-from** option for more information.

## 10.2. Restore

When an application developer runs the **rhc snapshot restore** command, OpenShift Enterprise restores the application from an archive in the following steps:

1. Prepares the application for restoration.
   - If the archive contains a Git repository, OpenShift Enterprise runs the **gear pre-receive** command.
   - If the archive does not contain a Git repository, OpenShift Enterprise runs the **gear stop** command.
2. Runs the **control pre-restore** command for each cartridge on the gear. This enables you to control the restoration of your cartridge, for example by deleting an old database dump.
3. Builds a list of file name changes to apply during the restoration from the **restore_transforms** entry in the **$cartridge_name/metadata/managed_files.yml** file for each cartridge on the gear.
4. Extracts the archive into the gear user's home directory, overwriting existing files and applying the file name changes listed in the **restore_transforms** entry in the **managed_files.yml** file.
5. Runs the **control post-restore** command for each cartridge on the gear. Use this script to load a database flat file into the running database.
6. Resumes the application.
   - If the archive contains a Git repository, OpenShift Enterprise runs the **gear postreceive** command.
   - If the archive does not contain a Git repository, OpenShift Enterprise runs the **gear start** command.
7. Will either stop or start the gear based on the state of the application before restoring.

### Restoring with Transformed File Names

Use the optional **restore_transforms** entry in the **$cartridge_name/metadata/managed_files.yml** file to provide scripts that transform file names when OpenShift Enterprise restores an application. This entry enables you to restore older snapshots to a newer cartridge with file name changes.

**Example 10.2. restore_transforms Entry**

```
restore_transforms:
- s|${OPENSHIFT_GEAR_NAME}/data|app-root/data|
```

OpenShift Enterprise uses the **tar** command when restoring a gear. See the **tar** man page **--transform** option for more information.

# Chapter 11. Upgrading Custom and Community Cartridges

The OpenShift Enterprise runtime contains a system for upgrading custom cartridges on a gear to the latest available version and for applying gear-level changes that affect cartridges.

The **`oo-admin-upgrade`** command on the broker host provides the command line interface for the upgrade system and can upgrade all the gears in an OpenShift Enterprise environment, all the gears on a node, or a single gear. This command queries the OpenShift Enterprise broker to determine the locations of the gears to migrate and uses MCollective calls to trigger the upgrade for a gear.

**Upgrade Process Overview**

1. Load the gear upgrade extension, if configured.
2. Inspect the gear state.
3. Run the gear extension's **`pre-upgrade`** script, if it exists.
4. Compute the upgrade itinerary for the gear.
5. If the itinerary contains an incompatible upgrade, stop the gear.
6. Upgrade the cartridges in the gear according to the itinerary.
7. Run the gear extension's **`post-upgrade`** script, if it exists.
8. If the itinerary contains an incompatible upgrade, restart and validate the gear.
9. Clean up after the upgrade by deleting pre-upgrade state and upgrade metadata.

## 11.1. Upgrade Itinerary

The upgrade process must be re-entrant; if it fails or times out, a subsequent upgrade operation must pick up where the last one left off. The upgrade itinerary stores information about which cartridges in a gear to upgrade and which type of upgrade to perform.

There are two types of cartridge upgrade processes: compatible and incompatible. The **`Compatible-Versions`** element in a cartridge's **`$cartridge_name/metadata/manifest.yml`** file determines whether the new version is compatible with a previous version. The main difference between the compatible and incompatible upgrade processes is that an incompatible cartridge's gear stops during an upgrade, while a compatible cartridge's gear continues to run.

**Upgrade Itinerary Configuration**

1. Read in the current **`IDENT`** of the cartridge.
2. Determine the name and software version of the cartridge in the cartridge repository; this provides the manifest for the latest version of the cartridge. If a manifest does not exist in the cartridge repository or does not include the software version, skip the cartridge.
3. If the latest manifest is for the same cartridge version as the version currently installed on the gear, skip the cartridge unless the **`ignore_cartridge_version`** parameter is set. If the **`ignore_cartridge_version`** parameter is set, record an incompatible upgrade for the cartridge in the itinerary.

4. If the latest manifest includes the current cartridge version in the **`Compatible-Versions`** element, record a compatible upgrade for the cartridge in the itinerary. Otherwise, record an incompatible upgrade for the cartridge in the itinerary.

## 11.2. Compatible Upgrades

If the upgrade itinerary records a compatible upgrade for a cartridge, OpenShift Enterprise uses the following process:

**Compatible Upgrade Process**

1. Overlay the new version of the cartridge on the gear.
2. Remove the files declared in the **`Processed-Templates`** element of the cartridge's **`managed-files.yml`**.
3. Unlock the cartridge directory.
4. Secure the cartridge directory.
5. Run the cartridge's **`upgrade`** script, if it exists.
6. Lock the cartridge directory.

## 11.3. Incompatible Upgrades

If the upgrade itinerary records an incompatible upgrade for a cartridge, OpenShift Enterprise uses the following process:

**Incompatible Upgrade Process**

1. Remove the files and directories declared in the **`Setup-Rewritten`** element of the cartridge's **`managed_files.yml`**.
2. Overlay the new version of the cartridge on the gear.
3. Unlock the cartridge directory.
4. Secure the cartridge directory.
5. Run the cartridge's **`upgrade`** script, if it exists.
6. Run the cartridge's **`setup`** script.
7. Process the cartridge's ERB templates.
8. Lock the cartridge directory.
9. Create new endpoints for the cartridge.
10. Connect the frontend.

## 11.4. Cartridge Upgrade Script

You can provide a cartridge **`upgrade`** script in the **`$cartridge_name/bin/`** directory to run during the upgrade process. The **`upgrade`** script enables you to perform actions during the upgrade process that the compatible or incompatible processes do not perform. If you provide an **`upgrade`** script, OpenShift Enterprise passes it the following arguments:

- The software version of the cartridge.
- The current cartridge version.
- The cartridge version being upgraded to.

A non-zero exit code from this script results in the upgrade operation failing.

# Chapter 12. Enabling Logshifter

Using **logshifter** enables automatic log rotation and consolidation across cartridges in a gear. In OpenShift Enterprise, cartridges can log to **Syslog** using **logshifter**. For more information about **logshifter**, see the **logshifter `README`** file.

When writing a cartridge control script, it is typical to spawn a runtime process such as **`java`** or **`mongod`**. To take advantage of **logshifter**, redirect the standard out (STDOUT) and standard error (STEDRR) streams of the process to the **`/usr/bin/logshifter`** file. For example, for a Java-based cartridge:

**Example 12.1. Redirecting Logs Using Logshifter:**

```
java ... |& /usr/bin/logshifter -tag my-cartridge &
```

Using this method, **`java`** is started in the background, and all output produced by the application is logged through **logshifter**. The ***`-tag`*** argument must be a string unique to the cartridge.

### Tips for PID Management

In the example above, the standard pipe operator is used from a shell script to redirect logs from the cartridge process to **`logshifter`**. This works well for programs which are capable of managing a PID file internally. However, for cartridges which bootstrap a process and cannot manage a PID file by itself, using a simple pipe operator can be problematic. When piping programs using a shell, the programs are typically started in parallel, rendering the **`$!`** variable unreliable for determining the PID of the cartridge process. For these cases, setting up a named pipe can allow the cartridge to use **`logshifter`** and also manage the PID of the process. The following example demonstrates the setup of a named pipe used by a **Java** cartridge which preserves the reliability of **`$!`** so the cartridge script can manage the PID file manually:

**Example 12.2. Setting Up Named Pipe:**

```
LOGPIPE=${OPENSHIFT_HOMEDIR}/app-root/runtime/logshifter-my-cartridge
rm -f $LOGPIPE && mkfifo $LOGPIPE

/usr/bin/logshifter -tag my-cartridge < $LOGPIPE &
java ... &> $LOGPIPE &
echo $! > $OPENSHIFT_MY_CARTRIDGE_DIR/my-cartridge.pid
```

Setting up the named pipe manually provides the most flexibility for managing both the **`logshifter`** and cartridge processes.

# Chapter 13. OpenShift Cartridge Reference

This chapter contains reference material for OpenShift Enterprise cartridges.

## 13.1. Cartridge Hierarchy

OpenShift cartridges use a hierarchy so that multiple cartridges can be collocated or combined on one gear. This hierarchy consists of a single, **`primary`** cartridge, along with a combination of **`embedded`** cartridges. The **`primary`** cartridge controls the application build life cycle, responds to scaling events, and provides external network accessibility. The **`embedded`** cartridges support the **`primary`** cartridge and provide additional capabilities to applications.

A good example of this hierarchy is the relationship between the **`jenkins`** cartridge and the **`jenkins-client`** cartridge. While the **`jenkins`** cartridge provides a fully functional browser based Jenkins service, the **`jenkins-client`** cartridge is embedded in web applications to offload application builds to an existing Jenkins service. Note that the **`jenkins-client`** cartridge by itself provides no value and must be combined with an existing **`primary`** cartridge.

## 13.2. Cartridge Directory Structure

The required directories of a cartridge must conform to a set structure or the cartridge can fail to function properly. You can add additional directories and files as needed to support the function of your cartridge.

- **`Required`** files must exist for minimal OpenShift Enterprise support of the cartridge.
- **`Discretionary`** files are recommended, but not necessary. For example, **`conf.d`** is the standard file where a web framework installs its **`httpd`** configuration.
- **`Optional`** files are not necessary. Use optional files to support additional cartridge functionality.

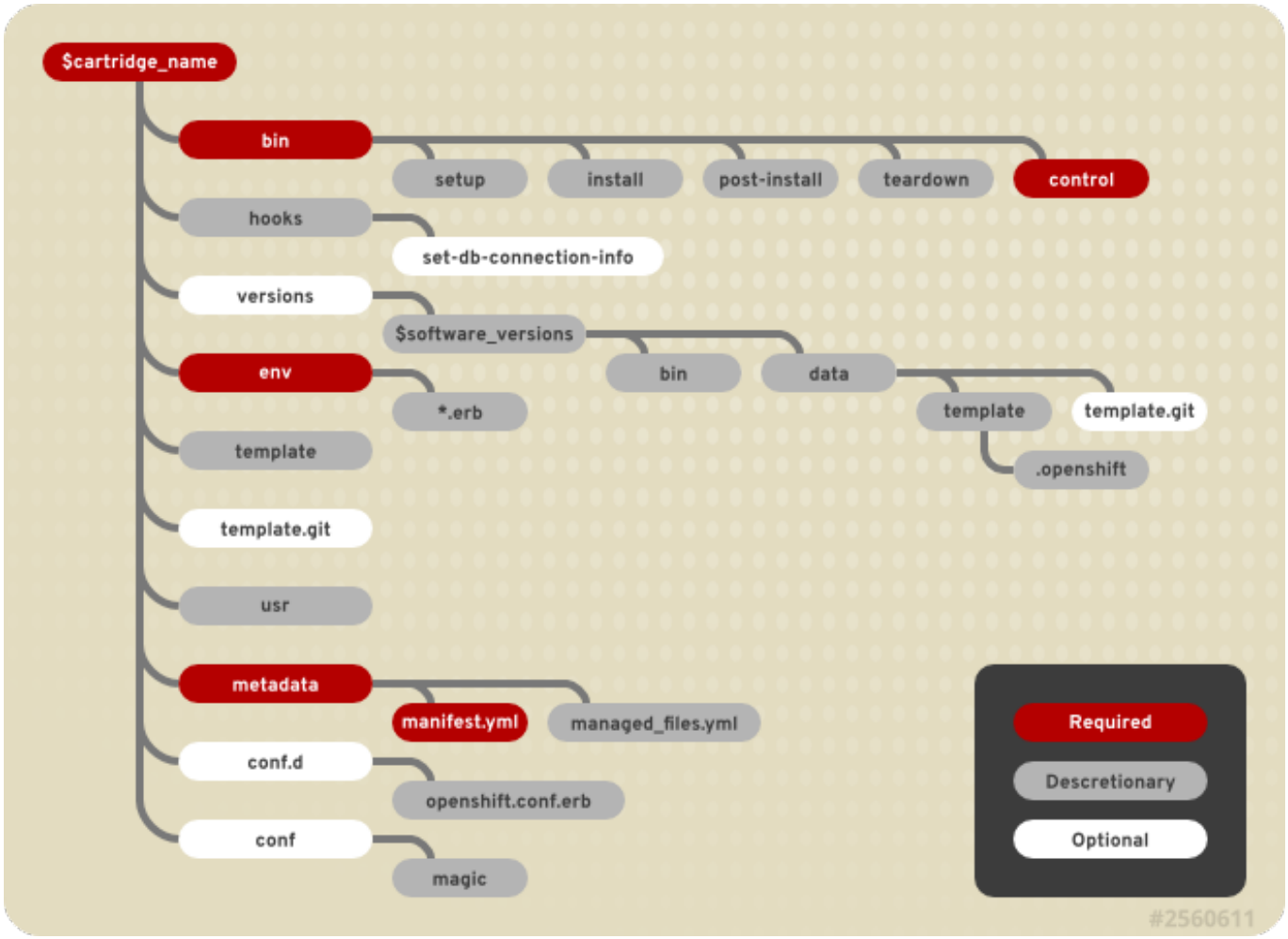



**Figure 13.1. Cartridge Directory Structure**

To support multiple software versions within one cartridge, create symbolic links between the **`$cartridge_name/bin/control`** file and the **`$cartridge_name/versions/$software_version/bin/control`** file. Alternatively, use the **`$cartridge_name/bin/control`** file as a shim to call the **`control`** file for the desired version.

When creating an instance of your cartridge for use by a gear, OpenShift Enterprise copies the files, links, and directories from the cartridge library, with the exception of the **`$cartridge_name/usr/`** directory. The **`$cartridge_name/usr/`** directory is symbolically linked to the gear's cartridge instance. This link enables all cartridge instances to share libraries and other data.

See Chapter 3, *Locking Cartridges* for details on customizing a cartridge instance.

## 13.3. Cartridge Metadata Elements

OpenShift Enterprise uses a **`manifest.yml`** file located in the **`$cartridge_name/metadata/`** directory to determine the features a cartridge requires. OpenShift Enterprise also uses data from the **`manifest.yml`** file to provide information about the cartridge to users.

**Example 13.1. manifest.yml File**

```
Name: PHP
Cartridge-Short-Name: PHP
Cartridge-Version: '1.0.1'
```

```
Compatible-Versions:
  - '1.0.1'
Cartridge-Vendor: redhat
Display-Name: PHP 5.4
Description: "PHP is a general-purpose server-side scripting language..."
Version: '5.4'
Versions:
  - '5.4'
License: "The PHP License, version 3.0"
License-Url: http://www.php.net/license/3_0.txt
Vendor: PHP Group
Categories:
  - service
  - php
  - web_framework
Website: http://www.php.net
Help-Topics:
  "Developer Center": https://openshift.redhat.com/community/developers
Cart-Data:
  - Key: OPENSHIFT_...
    Type: environment
    Description: "How environment variable should be used"
Provides:
  - php-5.4
  - "php"
Publishes:
  get-php-ini:
    Type: "FILESYSTEM:php-ini"
  publish-http-url:
    Type: "NET_TCP:httpd-proxy-info"
  publish-gear-endpoint:
    Type: "NET_TCP:gear-endpoint-info"
Subscribes:
  set-db-connection-info:
    Type: "NET_TCP:db:connection-info"
    Required: false
  set-nosql-db-connection-info:
    Type: "NET_TCP:nosqldb:connection-info"
    Required: false
  set-mysql-connection-info:
    Type: "NET_TCP:db:mysql"
    Required : false
  set-postgres-connection-info:
    Type: "NET_TCP:db:postgres"
    Required : false
  set-doc-url:
    Type: "STRING:urlpath"
    Required : false
Scaling:
  Min: 1
  Max: -1
Group-Overrides:
  - components:
    - php-5.4
    - web_proxy
Endpoints:
```

```
  - Private-IP-Name:   IP1
    Private-Port-Name: HTTP_PORT
    Private-Port:      8080
    Public-Port-Name:  PROXY_HTTP_PORT
    Mappings:
      - Frontend:      '/front'
        Backend:       '/back'
Additional-Control-Actions:
  - threaddump
```

### 13.3.1. Cartridge-Short-Name

OpenShift Enterprise creates several environment variables when installing a cartridge. The **`Cartridge-Short-Name`** element forms part of these environment variable names.

**Example 13.2. Cartridge-Short-Name Entry for a PHP cartridge**

```
Cartridge-Short-Name: PHP
```

Environment variables use **`PHP`** in their name:

```
OPENSHIFT_PHP_DIR
OPENSHIFT_PHP_IP
OPENSHIFT_PHP_PORT
OPENSHIFT_PHP_PROXY_PORT
```

### 13.3.2. Cartridge-Version

The **`Cartridge-Version`** element identifies the release version of a cartridge. The value uses the format:

```
<number>[.<number>[.<number>[...]]]

For example:

Cartridge-Version: '1.0.3'
```

When you publish a new version of a cartridge, OpenShift Enterprise uses the **`Cartridge-Version`** value to determine upgrade requirements for applications that use the cartridge. YAML treats **`number.number`** as a float but OpenShift Enterprise requires a string for this value, so the value must be enclosed in single quotes (').

### 13.3.3. Compatible-Versions

The **`Compatible-Versions`** element is a list of previous cartridge versions that are compatible with the current cartridge version.

**Example 13.3. Compatible-Versions Entry**

```
Compatible-Versions: ['1.0.1']
```

To be compatible with a previous version, the code changes in the current cartridge version must not require a restart of the cartridge or of an application using the cartridge.

If the previous cartridge version is not in the **Compatible-Versions** list when you update the cartridge to a new version, OpenShift Enterprise stops the cartridge, installs the new code, runs **setup**, and restarts the cartridge. This process results in a short amount of downtime for applications that use the cartridge.

### 13.3.4. Cartridge-Vendor

The **Cartridge-Vendor** element identifies the creator of a cartridge. OpenShift Enterprise uses this value to differentiate between similar cartridges installed on the system. You can use a company name or an individual identifier for this value.

**Example 13.4. Cartridge-Vendor Entry**

```
Cartridge-Vendor: redhat
```

### 13.3.5. Version

The **Version** element is the default version of the software packaged in the cartridge.

**Example 13.5. Version Entry**

```
Version: '5.3'
```

### 13.3.6. Versions

The **Versions** element is the list of software versions packaged in the cartridge.

**Example 13.6. Versions Entry**

```
Versions: ['5.3']
```

### 13.3.7. Categories

The **Categories** element is a list of classifications for a particular cartridge, and contains two distinct groups:

- **system** categories are special to the platform and influence the system behavior.
- **descriptive** categories are arbitrary classifications that improve the searching of cartridges in the Management Console and the client tools.

#### 13.3.7.1. System Categories

**system** categories contain the following subcategories:

- web_framework
- web_proxy
- service
- plugin
- embedded
- domain_scope

### Web Framework Category

The **web_framework** category describes cartridges that accept inbound HTTP, HTTPS, and WebSocket requests. SSL termination occurs at the platform layer before cartridge interaction. The original inbound protocol is passed to the cartridge using the **X-Forwarded-Proto** header. An application can have one cartridge from the **web_framework** category.

### Web Proxy Category

The **web_proxy** category describes cartridges that route web traffic to the application's gears. When a scalable application is created with a cartridge from the **web_framework** category, a **web_proxy** cartridge is automatically added to enable the auto scaling feature. Therefore, when a **web_framework** cartridge has to scale beyond a single gear, the **web_proxy** cartridge automatically routes to the endpoint defined by the **Public-Port-Name** with the **PROXY_PORT** value. The **web_proxy** cartridge is automatically updated over HTTP with routing rules of the new gears as they are added. An application can have one cartridge from the **web_proxy** category.

### Service Category

The **service** category describes add-on cartridges that are not based on HTTP, such as MySQL. The **service** category cartridges can scale independently, but may not be addressable outside of the platform. Therefore, OpenShift Enterprise applications must have at least one **web_framework** category cartridge so that the application's DNS registration contains at least one addressable HTTP endpoint. However, most applications consist of a **web_framework** category cartridge and other cartridges from the **service** category. Therefore, using the **service** category to classify a cartridge, such as MySQL, installs the cartridge on a separate gear from that of the **web_framework** cartridge. This allows both cartridges to scale independently.

### Embedded Category

The **embedded** category describes cartridges that are always collocated or installed with any other **primary** cartridge in non-scalable applications. For example, the Jenkins client cartridge can be combined with any web application cartridge to offload the builds to a Jenkins service.

### Plugin Category

The **plugin** category is similar to the **embedded** category, but for scalable applications. It describes cartridges that can be collocated with other cartridges in scalable applications. The **plugin** category uses defined **Group-Overrides** to determine the collocation between cartridges. For example, the **Group-Overrides** can specify that a Cron cartridge must be collocated with the **web_framework** category cartridge.

#### Domain Scope Category

The **domain_scope** category describes cartridges that can only have a single instance within a domain. For example, the Jenkins server cartridge contains the **domain_scope** category to ensure that there is only one Jenkins server application within an entire domain. The Jenkins client cartridge is embedded in all other applications to enable builds that are handled by the Jenkins server.

### 13.3.7.2. Descriptive Categories

The **descriptive** categories are mostly used in the OpenShift Enterprise Management Console and the client tools to improve the overall user experience. In the Management Console, the **descriptive** categories are used as tags that allow users to search and filter the available cartridges.

When using the client tools, the **descriptive** categories are used to apply matching logic to cartridge operations. For example, if a user runs the **rhc add-cartridge php** command, the **descriptive** categories are searched along with the names of the cartridges.

## 13.3.8. Group-Overrides

By default, each cartridge in a scalable application resides on its own gear within its own group instance. **Group-Overrides** can be used when you wish to have two cartridges located on the same set of gears. For example, if you create a **cron** cartridge and wish to collocate that with a **web_framework** category cartridge, you can do so as shown in the following example.

**Example 13.7. Group-Overrides with cron and web_framework Cartridges**

```
Group-Overrides:
- components:
  - web_framework
  - cron
```

In cases where you wish to collocate a **web_framework** category cartridge with a **web_proxy** category cartridge, you can do so as shown in the following example.

**Example 13.8. Group-Overrides with web_framework and web_proxy Cartridges**

```
Group-Overrides:
- components:
  - web_proxy
  - web_framework
```

## 13.3.9. Scaling

When a cartridge is added to a scalable application, the **Min** and **Max** parameters define the scaling limits for that cartridge. If both the **Min** and **Max** values are set to 1, this means that the cartridge cannot scale. If the **Max** value is set to -1, the cartridge can scale up to the user's maximum gear limit. These limits are applicable for both automatic and manual scaling of cartridges.

Note that when using **Group-Overrides** to collocate two or more scalable cartridges, the scaling limits of both cartridges must match. However, there may be cases where this limitation may not be ideal; for example, a **web_proxy** category cartridge collocated with a **web_framework** category cartridge. In such a

case, it is not recommended to have the **web_proxy** cartridge be located on every gear that holds the **web_framework** cartridge. The **Multiplier** parameter allows you to place a cartridge only on certain gears within a group instance, rather than all of them. For example, if the **Multiplier** is set to 3, every third gear within the group instance gets the cartridge installed on it. If it is set to 1, then the cartridge gets installed on all gears within the group instance.

### 13.3.10. Source-Url

The **Source-Url** element is the location from which OpenShift Enterprise downloads cartridge files during application creation.

**Table 13.1. Supported Source Schemes**

| Scheme | Method | Expected Inputs |
|---|---|---|
| git | clone | git repo |
| https | GET | zip, tar, tag.gz, tgz |
| http | GET | zip, tar, tag.gz, tgz |
| file | file copy | cartridge directory tree |

**Example 13.9. Source-Url Entry**

```
Source-Url: https://github.com/example/killer-cartridge.git
Source-Url: git://github.com/chrisk/fakeweb.git
Source-Url: https:://www.example.com/killer-cartridge.zip
Source-Url: https://github.com/example/killer-cartridge/archive/master.zip
```

### 13.3.11. Source-Md5

The **Source-Md5** element is an MD5 digest. If OpenShift Enterprise downloads a cartridge using a non-Git scheme, it verifies the downloaded file against this MD5 digest.

**Example 13.10. Source-Md5 Entry**

```
Source-Md5: 835ed97b00a61f0dae2e2b7a75c672db
```

### 13.3.12. Additional-Control-Actions

The **Additional-Control-Actions** element is a list of optional actions a cartridge supports. OpenShift Enterprise can only call optional actions if they are included in this element.

**Example 13.11. Additional-Control-Actions Entry**

```
Additional-Control-Actions:
  - threaddump
```

### 13.3.13. Endpoints

See Chapter 4, *Exposing Services*.

## 13.4. Example openshift.conf.erb File

**httpd** is a common base for OpenShift Enterprise cartridges. You can use this example **conf.d/openshift.conf.erb** file as a starting point for writing a cartridge based on **httpd**.

```
ServerRoot "<%= ENV['OPENSHIFT_HOMEDIR'] + "/ruby-1.8" %>"
DocumentRoot "<%= ENV['OPENSHIFT_REPO_DIR'] + "/public" %>"
Listen <%= ENV['OPENSHIFT_RUBY_IP'] + ':' + ENV['OPENSHIFT_RUBY_PORT'] %>
User <%= ENV['OPENSHIFT_GEAR_UUID'] %>
Group <%= ENV['OPENSHIFT_GEAR_UUID'] %>

ErrorLog "|/usr/sbin/rotatelogs <%= ENV['OPENSHIFT_HOMEDIR']%>/ruby-
1.8/logs/error_log-%Y%m%d-%H%M%S-%Z 86400"
CustomLog "|/usr/sbin/rotatelogs <%=
ENV['OPENSHIFT_HOMEDIR']%>/logs/access_log-%Y%m%d-%H%M%S-%Z 86400" combined

PassengerUser <%= ENV['OPENSHIFT_GEAR_UUID'] %>
PassengerPreStart http://<%= ENV['OPENSHIFT_RUBY_IP'] + ':' +
ENV['OPENSHIFT_RUBY_PORT'] %>/
PassengerSpawnIPAddress <%= ENV['OPENSHIFT_RUBY_IP'] %>
PassengerUseGlobalQueue off
<Directory <%= ENV['OPENSHIFT_REPO_DIR]%>/public>
  AllowOverride all
  Options -MultiViews
</Directory>
```

# Appendix A. Revision History

| Revision 2.1-3 | Mon Sep 15 2014 | Bilhar Aulakh |
|---|---|---|

BZ 1114617: Updated Section 6.1, “Embedded Ruby (ERB) Processing” with correct file names.

| Revision 2.1-2 | Thu Jun 26 2014 | Julie Wu |
|---|---|---|

Added Chapter 12, *Enabling Logshifter*.
Updated Section 6.7, “metrics Script” with 2.1 release callout.

| Revision 2.1-1 | Fri May 16 2014 | Julie Wu |
|---|---|---|

OpenShift Enterprise 2.1 release.
BZ 1097056: Updated Section 4.1, “TCP Endpoints”, Section 4.2, “TCP Endpoint Example”, and Section 13.3, “Cartridge Metadata Elements”.
Added Section 6.7, “metrics Script”.
Added Section 4.4, “Enabling Custom Paths for Websockets”.
Updated Section 10.2, “Restore” and Section 10.1, “Snapshot”.

| Revision 2.0-0 | Mon Dec 9 2013 | Bilhar Aulakh |
|---|---|---|

OpenShift Enterprise 2.0 release.
Add section on cartridge categories.
Add section on cartridge hierarchy.
Update default lifecycle phases.
Add section on archiving applications.
Add section on binary deployment.