YOUR BRAND NAME HERE
your contact email here | your contact phone here
your website here

Project Developer Guide

Version 1.0 International English (en), January 2023

Table of Contents

1 . Prologue

First of all, thank you for selecting and using my AutomataCI product. This document is a developer-specific guidance for kick-starting or contributing to a repository supported by AutomataCI. In case of any inquiry, please feel free to contact us at:

1. hollowaykeanho@gmail.com OR hello@hollowaykeanho.com
2. https://github.com/orgs/ChewKeanHo/discussions

IMPORTANT NOTE: Please change the content of this section. TQ

2 . Developer's Quick-Start

This section covers a quick re-cap for the experienced developers developing and maintaining the project without needing to go through the entire specification.

For new developers, please do go through the engineering specifications in order to understand how the Project operates and manages.

The steps are prepared in sequences.

2.1 . Git Clone the Repository

To obtain a local copy, simply use git to clone the repo:

```
$ git clone <project_url>
$ cd <project>
```

2.2 . CONFIG.toml defines The Repository Filesystem

Please at least read through the `CONFIG.toml` configuration file to have a fresh re-cap what directories are used for what purposes.

2.3 . To Setup The Repository for Development

To quickly setup the project repository simply:

```
$ ./ci.cmd setup
```

Upon completion, your repository is ready for development.

2.4 . To Start A Development

To start the development after setting up, simply:

```
$ ./ci.cmd start
```

Upon completion, your terminal should be ready for development.

2.5 . To Execute The Test Cycle

To run a test cycle, simply:

```
$ ./ci.cmd test
```

Upon completion, please check your `{PROJECT_PATH_ROOT}/{PROJECT_PATH_LOG}` for the test report and coverage heatmap if available.

2.6 . To Build The Products

To build the production-ready product, simply:

```
$ ./ci.cmd build
```

Upon completion, please check your `{PROJECT_PATH_ROOT}/{PROJECT_PATH_BUILD}` for the built files.

2.7 . To Compose All Documents

To compose all the documentations, simply execute the following locally:

```
$ ./ci.cmd compose
```

Upon completion, TODO: pending development.

2.8 . To Package Products

To package the product, simply execute the following locally (in case secret keys and certs are involved):

```
$ ./ci.cmd package
```

Upon completion, please check your `{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}` for all successfully packed packages.

2.9 . To Release The Packages

To release the product, simply execute the following locally:

```
$ ./ci.cmd release
```

Upon completion, you may check the published updates in the publishers' store.

2.10 . To Publish the Composed Documents

To publish the composed documentations, simply execute the following locally:

```
$ ./ci.cmd publish
```

Upon completion, TODO: pending development.

3 . Granted Licenses

This product is licensed under one or many of the software licenses listed in a separate and accompanied document called `LICENSES.pdf` or `LICENSES.md` depending on its encoded format.

IMPORTANT NOTE: Please change the content of this section. TQ

3.1 . AutomataCI Source Codes

The AutomataCI source codes are under the Apache 2.0 Open Source License as inbound licenses. Usually, it is prohibited for redistribute due to business priorities.

4 . Engineering Specification

The product engineering specification are defined here.

IMPORTANT NOTE: Please change the content of this section. TQ

5 . Infrastructure – AutomataCI

This section is about the specification of the AutomataCI stewarding the repository semi-automatically for long-term sustainable maintenance purposes.

5.1 . Why Another Automation CI

AutomataCI was specifically built to counter supply-chain threat encountered since Year 2022 across the Internet service providers. A white paper is available for detailing of the incident and for case study education purposes is available here: https://doi.org/10.5281/zenodo.6815013.

Ever-since the post Covid-19 pandemic, a lot of CI service providers are drastically changing their business offering to the extent of extorting their customers to either pay a very high price or close the entire project operation down. In response to such threat, ZORALab's Monteur (https://github.com/zoralab/monteur) was first created to remove such a threat but it has its own flaws dealing with various OSes native functions.

Hence, AutomataCI was created iteratively to resolve ZORALab's Monteur weakness, allowing a project repository to operate without depending entirely on ZORALab's Monteur's executables.

5.2 . The AutomataCI Mantra

The sole reason for deploying a CI from the get-go is to make sure the project life-cycle can be carried out consistently anywhere and everywhere the project is configured for, remotely via cloud or even locally. It facilitates heavily resistances and resilience to market changes without hampering the product development and production ecosystem.

Unlike other CI models, **AutomataCI favors "semi-automatic" approach** where the automation **can also be manually controlled stages by stages while still facilitating full automation**. Moreover, it allows localized CI run instead of getting involved with vendor locked-in and expose the project for extortion legally. If a CI service provider turns sour, one can easily switch to another.

5.3 . Technological Requirements

To be seamlessly compatible with the OS natively, the entire AutomataCI is created using only POSIX compliant shell (not BASH) scripts and Windows' PowerShell scripts. At its root, to make the interfaces unified, the Polygot Script (https://github.com/ChewKeanHo/PolygotScript) is used where the POSIX shell and Batch script are unified into a single file called `ci.cmd`.

Generally speaking, you only need the following:

1. **POSIX Shell Scripting (not BASH)** – for all UNIX OSes including Apple MacOS; AND
2. **Windows PowerShell** – for Windows OS.

Keep in mind that, although the `ci.cmd` is using Batch scripting, you do not need Batch script anymore. If you're still using one, you're doing things the wrong way at least in the AutomataCI perceptive (and it is way too arcane for CI responsibilities anyway).

Both of these technologies are inherently known to any developer as they're just the same commands typed into the terminal. The difference is that AutomataCI captures them in a script and turn them into reusable tools. Therefore, it has less learning curves.

5.3.1 . External CI Services

To counter the external CI service providers' sudden threatening changes, their interface with the project is **to strictly call the AutomataCI's APIs the same way you're using it locally**. If available, you can view the `.github/workflow` GitHub CI's workflow recipes to see how the integration happens. That way, the project's critical processes are not externally affected and if they're in-fact threatening, the switching to another CI service provider is seamless and easy or one can operate locally without impacting the production schedule.

Due to the fact that CI is an important life-support system for the project, **you're strongly advised not to use any vendor-specific API or functionalities. Anything AutomataCI can't do locally means it is vendor locked-in**. The more such you use, the more entangled you are; which also means the more painful for you to do immediate migration when threat suddenly appear.

5.4 . Filesystem

The AutomataCI requires at least 3 important elements to operate properly:

1. The `ci.cmd` – The Polygot script that unifies all OSes start point meant for you to trigger a CI job.
2. The `automataCI` libraries – A directory housing all the AutomataCI job recipes and function services.
3. The **CONFIG.toml** repo config file – A simple TOML formatted config file that provides the repository's critical parameters for AutomataCI to operate and manage with.

There are specific filesystem used by AutomataCI defined in the **CONFIG.toml** file. To avoid duplication, you should go through that TOML file and understand what are being used for the project in order to avoid any conflicts. By default, the following file structures are defined:

automataCI/	➞ house the projects' CI automation scripts.
automataCI/services	➞ house tested and pre-built CI automation functions.
bin/	➞ default build output directory.
pkg/	➞ default package output directory.
resources/	➞ housing all indirect raw materials and assets.
resources/packages	➞ housing all packages control template files.
resources/icons	➞ housing all graphics and icon files.
resources/licenses	➞ housing all project licensing generative documents.
resources/docs	➞ housing all project's document generators.
resources/changelog	➞ housing all project's changelog entries data.
src/	➞ house actual source codes (base directory).
src/.ci/	➞ house source codes technology-specific CI job recipes.
tools/	➞ default tooling (e.g. programming language's bin/* executables).
tmp/	➞ default temporary workspace.
CONFIG.toml	➞ configure project's settings data for AutomataCI.
ci.cmd	➞ CI start point.
.git	➞ Git version control configuration directory.

5.4.1 . File Extensions

AutomataCI uses the default file extensions without any new invention. Basically they are:

1. **.cmd** – for batch and POSIX shell polygot script only.
2. **.sh** – for all POSIX shell scripts only.
3. **.ps1** – for all PowerShell scripts only.
4. **.toml** – for CONFIG.toml file only.

5.4.2 . Filename Convention

AutomataCI uses:

1. underscore (_) for context switching; AND
2. dash (-) for same context but different aspect separations; AND
3. No space is allowed.

Each job and service script are accompanied by a system specific naming convention complying to this pattern:

{PURPOSE}_{PROJECT_OS}-{PROJECT_ARCH}.{EXTENSION}

For example, for a setup job recipe, the filename can be any of the following:

1. **setup_windows-amd64.ps1** – PowerShell script for Windows OS, with amd64 CPU only.
2. **setup_windows-any.ps1** – PowerShell script for Windows OS with any CPU types.
3. **setup_unix-amd64.sh** – POSIX shell script for UNIX OS (Linux, Hurd, and Apple MacOS) with amd64 CPU only.
4. **setup_unix-any.sh** – POSIX shell script for UNIX OS with any CPU types.
5. **setup_darwin-any.sh** – POSIX shell script for Apple MacOS only with any CPU types.

In any cases, if you know the content of the script does not rely on specific CPU, you generally just name it as:

1. **{purpose}_unix-any.sh**
2. **{purpose}_windows-any.ps1**

and place them next to each other will do.

5.5 . CI Jobs

This section covers all the CI jobs. AutomataCI employs a linear story-line style of job executions. In each job, the developer can deploy concurrent or parallel executions as the supporting OS permits and as long as it makes sense. The default is usually synchronous since CI demands clarity and high accuracy over speed when it comes to generating a proper customer-usable products.

The CI jobs can be customized without modifying the critical recipe files by detecting a **`.ci/{purpose}_{PROJECT_OS}-any.{EXTENSION}`** job recipe file inside your **`{PROJECT_PATH_ROOT}/{PROJECT_PATH_SOURCE}`** directory. When detected, the customized job recipe shall be executed instead of the default one.

Basically, the execution sequences are as follow:

1. user only triggers the CI jobs via **`ci.cmd`**; AND
2. Then the **ci.cmd** sorts out the platform specific data and shall calls the CI job recipes housed in the first level of **`automataCI`** directory (e.g. *`automataCI/setup_windows_any.ps1`* for setup job operating in Windows OS regardless of *amd64* or *arm64* CPU architecture); AND
3. CI job recipes detect customized job recipe and execute it if available or otherwise, execute its known default tasks.

Example, say there is a *`setup_windows-any.ps1`* job recipe in *`src/.ci/`* directory, assuming `PROJECT_PATH_SOURCE` is set to "`src`" directory in the `CONFIG.toml` file, the execution sequence shall be:

1. User executed "**`./ci.cmd setup`**" command in Windows OS with amd64 CPU.
2. **`ci.cmd`** seeks out **`automataCI/setup_windows-any.ps1`** job recipe and execute it.
3. **`automataCI/setup_windows-any.ps1`** detected ***`src/.ci/setup_windows-any.ps1`*** custom job recipe and execute it instead of its default executions.

5.5.1 . Setup

Setup job recipe is responsible for setting up the required tooling (e.g. programming languages' compilers, build tools, test tools, test coverage heat-map tools, and etc) by either downloading from the Internet or the Intranet safely.

Example, for Go programming language, it's downloading the tar.gz engine archive and unpack it to the `PROJECT_PATH_TOOLS/go-engine` directory.

5.5.1.1 . Operating Parameters

This job place it output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/{brand}[-engine]

5.5.2 . Start

Start job recipe is responsible for setting up the development environment in the terminal for actual development.

Example, for Go programming language, it's to setup the `GOPATH`, `GOBIN`, and etc against the localized Go compiler so that the developer can continue the Go project development.

Another example: for Python programming language, it is to initialize the venv virtual environment.

5.5.2.1 . Operating Parameters

This job takes its no special input path.

It generates output files and directories in the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/{brand}[-engine]

5.5.3 . Prepare

Prepare job is responsible to prepare the repository up to a designated version and configurations. This includes managing your project dependencies and generating the required version files.

Example, for Go programming language, it's executing the `go get` command, clean up, and generating the VERSION file.

Another example: for Python programming language, it's executing the `pip install -r requirements.txt` command.

5.5.3.1 . Operating Parameters

This job takes the following as its output:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TEMP}/

5.5.4 . Test

Test job is responsible for initiating the project test cycle and execution such as but not limited to unit tests, integration test, and etc alongside test coverage heat-mapping.

Example, for Go programming language, it executes the "`go test`" command alongside generating the source code test coverage heatmap to empower the developer/tester to analyze and test effectively.

Another example, for Python programming language, it's executing `unittest` alongside `coverage` module.

5.5.4.1 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_SOURCE}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_LOG}/

5.5.5 . Build

Build job is responsible for building the project production-ready output such as but not limited to building the executable binary without any debugging symbols, and assembling its necessary dependencies in necessary locations for package CI job later.

As the distribution ecosystem are moving towards server containerization and ease-of-use cases, **it's always advisable to produce a single binary executable and refrain from requesting your customers to sort out your dependencies**. It's your job, not theirs.

To speed up the process, developers can deploy concurrent or parallel executions facilitated by AutomataCI existing services OS library (explain later).

Also, please do note that the output of the executable file shall always comply to the following naming convention to make sure the Package job default tasks executions are fully compatible:

`{PROJECT_SKU}_{OS}-{ARCH}[.exe]`

Example: for Go programming language, say the PROJECT_SKU is "`myproc`" and is built against dragonfly, linux, openbsd, and windows OSes for amd64, arm64 CPUs, the list of output executable shall be:

1. `myproc_dragonfly-amd64`
2. `myproc_dragonfly-arm64`
3. `myproc_linux-amd64`
4. `myproc_linux-arm64`
5. `myproc_openbsd-amd64`
6. `myproc_openbsd-arm64`
7. `myproc_windows-amd64.exe`
8. `myproc_windows-arm64.exe`

Another example: for Python programming language, based on the above example, the build job shall execute `pyinstaller` to build a single binary of the project against the build system's OS and CPU architecture yielding the same naming convention as above.

5.5.5.1 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_SOURCE}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_BUILD}/

5.5.5.2 . Python

For building the binary that are coded using Python Programming Language, 3 Python related dependencies are required to compile the product into a single statically linked binary:

1. `pyinstaller`
2. `patchelf-wrapper` (linux-amd64 only)
3. `staticx` (linux-amd64 only)

The build process is as follows:

1. Product created from `main.py`.
2. Compile into single binary (partially static-linked) using `pyinstaller` with `--onefile` argument.
3. For `linux-amd64` variant, convert to full static-linked binary using `staticx`
4. final single-binary output.

These dependencies can be automatically installed using pip command, as such in the `requirements.txt` file:

```
# Please add your dependencies here



# CI required libraries. Please DO NOT remove. TQ!
coverage
pyinstaller
patchelf-wrapper
staticx
wheel
twine
```

5.5.5.2.1 . Customization

TBD

5.5.6 . Package

Package job is responsible for packaging the built binaries into the industrial known distribution channels such as but not limited to Windows Store, Debian APT `.deb` ecosystem, Red Hat's DEF `.rpm` ecosystem, Red Hat's Flatpak ecosystem, Apple's Brew ecosystem, CI friendly `tar.xz` or `.zip` archives ecosystem, etc that has default security protocols and with verifiable integrity.

This is a special job where instead of being override by a custom ci job recipe, the custom ci job recipe supplies the required content assembly functions instead.

The default package detects and validates all build binary based on the following naming convention in the `PROJECT_PATH_ROOT/PROJECT_PATH_BUILD` (defined in `CONFIG.toml`) directory:

```
{PROJECT_SKU}[-src]_{OS}-{ARCH}[.exe]
```

and package it based on the packager's availability in the CI host OS system (e.g. in Windows OS, packing `.deb` and `.rpm` are impossible as the packaging tools are unavailable and are incompatible). The minimum packaging output would be the `.tar.xz` and `.zip` archive files.

All successfully packed packages are housed in the `PROJECT_PATH_ROOT/PROJECT_PATH_PKG` (defined in `CONFIG.toml`) ready for release.

5.5.6.1 . Cryptography Requirements

Do note that some ecosystems require cryptography implementations such as but not limited to GPG signing for `.deb` and `.rpm` package types. If there are such a need, **it is always advisable to assemble all the built binary files in the right location and package it locally rather than relying on 3rd-party CI service provider**.

This is to protect the cryptography private keys from risking being exposed out (via 3rd-party service providers' contractors indirectly or security vulnerabilities) and always remain as secrets to your side.

5.5.6.2 . Special Custom Implementations

Unlike all other jobs, Package Job recipe **requires a compulsory custom CI job recipe** to supply the required package content assembly functions. Each of these function's specification are detailed in their respective sub-sections.

5.5.6.3 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_BUILD}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}/

5.5.6.4 . Archive Packages (.tar.xz|.zip)

5.5.6.5 . Debian Package (.deb)

AutomataCI developed its own package compiler based on the Debian Package specifications listed here (https://www.debian.org/doc/debian-policy/index.html) and here (https://manpages.debian.org/unstable/dpkg-dev/deb.5.en.html) to curb its massive-size and extremely complex default builder problem . Although AutomataCI uses its own compiler, the output shall always be compliant with upstream. Hence, you're required to learn through the specifications (at least binary package) shown above before proceeding to construct the job recipe.

AutomataCI prioritizes binary package build since the source codes are usually distributed using git control ecosystem. The removes any duplication related to the project and focus on customer delivery instead.

5.5.6.5.1 . Supported Platform

AutomataCI has a built-in available checking function that will check a given output target and host's dependencies' availability before proceeding. The table below indicates the supporting nature across known OS in the market.

UNIX (Debian based)	UNIX (Red Hat based)	MacOS	Windows
Supported	**???**	???	**Not Supported**

Not that should the host OS do not offer the required programs for the compilation, the compiler shall only issue a notice and skip its process.

Windows OS is not supported mainly due to the unique and required 'ar' archive program which is solely available for UNIX's coreutils package only. Until Microsoft provides such niche facility, it shall remain as it is.

5.5.6.5.2 . Content Assembling Function

The content assembling function is:

```
PACKAGE::assemble_deb_content() {
     …
}
```

Since Windows do not support `.deb` by default, there is no Windows counterparts.

In this function, the package Job shall pass in the following positional parameters:

```
__target="$1"
__directory="$2"
__target_name="$3"
__target_os="$4"
__target_arch="$5"
```

The `$__directory` variable should point to the workspace directory containing 2 important directories: `control/` and `data/`. The objective is to assemble the "to be installed" file structure in the `data/` directory and assemble any maintainer scripts (if needed) in the `control/` directory.

For example, the given `$__target` variable that is pointing to the currently detected and built binary program is usually being copied to `data/usr/local/bin` directory for compiling a binary package. This means that the package shall the target program into /usr/local/bin/ when the package is installed by the customer.

AutomataCI provides the ability to overrides any existing required files (see below). If these files are absent, AutomataCI shall generate one using its generator functions. Be warned that creating these required files can be a cumbersome effort (due to its steep technical debt). Hence, it is recommended to just focus on constructing the package's data path and leave the rest of the required files to the AutomataCI generative function.

If the function is unused, simply supply a single line with "`return 0`" is suffice to inform the shell that its does nothing.

5.5.6.5.2.1 . Required Files

As specified by Debian engineering specifications, there are 5 known required files:

1. `data/usr/share/docs/${PROJECT_SKU}/changelog.gz` **OR** `data/usr/local/share/docs/${PROJECT_SKU}/changelog.gz`
2. `data/usr/share/docs/${PROJECT_SKU}/copyright.gz` **OR** `data/usr/local/share/docs/${PROJECT_SKU}/copyright.gz`
3. `data/usr/share/man/man1/${PROJECT_SKU}.1.gz` **OR** `data/usr/local/share/man/man1/${PROJECT_SKU}.1.gz`
4. `control/md5sum`
5. `control/control`

These files follow strict format and content as specified in the Debian manual (especially `control/control` file).

To ensure consistency is secured and to prevent introducing any unnecessary difficulty, AutomataCI will generate these files when they're not being manually overridden during the content assembly function execution.

5.5.6.5.2.2 . Maintainers' Scripts

AutomataCI provides the ability for assembling the optional maintainers' scripts. They must be permitted for execution and are housed under `control/` directory, such that:

1. `control/preinst`
2. `control/postinst`
3. `control/prerm`
4. `control/postrm`

They are generally used for emergency patching, services (e.g. systemd, cron, nginx, etc) setup, and critical control during installation and removal. Their optional nature means you only assemble the scripts that you need and not using all of them is completely fine.

When in doubt, use `post[ACTION]` scripts.

5.5.6.5.3 . Collaborating With Automation – the `copyright.gz` file

AutomataCI constructs the `copyright.gz` file by generating the license stanza and then appending the copyright text file located here:

`{PROJECT_PATH_RESOURCES}/licenses/deb-copyright`

You should construct the license file as it is. Keep in mind that this file is heavily governed by Debian Policy Manual and you should at least go through the specification for binary package described here:

https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/

REMINDER: you only need to generate the body of the file as the automation will generate the license stanza automatically. If you need to modify the process, consider overriding the output manually.

Here's an example:

```
Files: automataCI/*, ci.cmd
Copyright: 2023 (Holloway) Chew, Kean Ho <hollowaykeanho@gmail.com>
License: Apache-2.0
 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at
 .
 http://www.apache.org/licenses/LICENSE-2.0
 .
 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.

Files: *
Copyright: {{ YEAR }} {{ YOUR_NAME_HERE }} <{{ YOUR_EMAIL_HERE }}>
License: {{ YOUR_SPDX_LICENSE_TAG_HERE }}
 {{ LICENSE_NOTICE }}
```

5.5.6.5.4 . Distributed Source Code Package

To create a source code package, simply place an empty file with the name complying to the following pattern:

```
{PROJECT_SKU}-src_{OS}-{ARCH}
```

This triggers the package job to recognize it as a target and you can assemble the `data/` path directly. The place-holding file can be created in the Build job recipe phase.

Keep in mind that by doing so, the `$PROJECT_SKU` value used in the automation shall automatically add a "`-src`" suffix for avoiding conflict with the binary package counterpart when both forms are distributed simultaneously.

Debian requires the OS and ARCH to be specific so the "any" ominous value is not available. Hence, please consider which OS and ARCH you wish to facilitate the development in order to restrict the development environment boundaries.

5.5.6.5.5 . Collaborating With Automation – Project Description Data

AutomataCI requires the long description data for generating the DEB's control file's `Description:` long data field facilitated by:

```
${__PROJECT_PATH_RESOURCES}/packages/DESCRIPTION.txt
```

Hence, please update the data there for consistencies across all package ecosystems.

5.5.6.5.6 . Testing Packaged DEB's Health

To test the packaged DEB's health, simply use the following command:

```
$ dpkg-deb --contents <package>.deb
$ dpkg-deb --info <package>.deb
```

If something goes wrong, `dpkg-deb` will report out for you.

5.5.6.6 . Red Hat Package (.rpm)

AutomataCI supported Red Hat native package known as “RPM” using their supplied development toolkit on supported platforms. It’s a UNIX (excluding MacOS) exclusive package especially operating on Red Hat based operating system such as but not limited to Fedora, CentOS, and etc. AutomataCI employs the following specifications provided by Red Hat to perform the RPM packaging accurately:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. http://ftp.rpm.org/api/4.4.2.2/specfile.html
3. https://developers.redhat.com/blog/2019/03/18/rpm-packaging-guide-creating-rpm
4. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#rpm-packages_packaging-software
5. https://stackoverflow.com/questions/15055841/how-to-create-spec-file-rpm
6. https://stackoverflow.com/questions/27862771/how-to-produce-platform-specific-and-platform-independent-rpm-subpackages-from-o
7. https://unix.stackexchange.com/questions/553169/rpmbuild-isnt-using-the-current-working-directory-instead-using-users-home

AutomataCI prioritizes binary package build since the source codes are usually distributed using git control ecosystem. This removes any duplication related to the project and focus on customer delivery instead.

5.5.6.6.1 . Supported Platform

AutomataCI can only build rpm output in the following build environment:

UNIX (Debian based)	UNIX (Red Hat based)	MacOS	Windows
Supported	**???**	**???**	**Not Supported**

Not that should the host OS do not offer the required programs for the compilation, the compiler shall only issue a notice and skip its process.

Windows OS is not supported mainly due to the absent of `rpmbuild` and `rpmsign` build tools.

5.5.6.6.2 . Content Assembly Function

The content assembling function is:

```
PACKAGE::assemble_rpm_content() {
     …
}
```

Since Windows do not support `.rpm` by default, there is no Windows counterparts.

In this function, the package Job shall pass in the following positional parameters:

```
__target="$1"
__directory="$2"
__target_name="$3"
__target_os="$4"
__target_arch="$5"
```

The `$__directory` variable should point to the workspace directory containing 2 important directories: BUILD/ and SPECS/. The objective is to assemble all the “to be installed” file structure in the BUILD/ directory and then assemble the spec file fragments in the `$__directory` location.

For example, the given `$__target` variable that is pointing to the currently detected and built binary program is usually being copied to `${__directory}/BUILD` directory. Then, to spin the required spec file fragment, simply use the printout to create them like:

```
        # generate AutomataCI's required RPM spec instructions (INSTALL)
        printf -- "\
install --directory %%{buildroot}/usr/local/bin
install -m 0755 ${PROJECT_SKU} %%{buildroot}/usr/local/bin

install --directory %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/
install -m 644 copyright %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/

install --directory %%{buildroot}/usr/local/share/man/man1/
install -m 644 ${PROJECT_SKU}.1.gz %%{buildroot}/usr/local/share/man/man1/
" >> "${__directory}/SPEC_INSTALL"
```

5.5.6.6.3 . Required Files

AutomataCI requires the following files to perform a successful build:

1. `${__directory}/SPEC_INSTALL`
2. `${__directory}/SPEC_FILES`

The content of the `${__directory}/SPEC_INSTALL` file is the `%install` commands that will be copied over during AutomataCI's automatic spec file generation. You're required to provide the instructions (without the `%install` stanza) in the content assembly function An example would be:

```
        # generate AutomataCI's required RPM spec instructions (INSTALL)
        printf -- "\
install --directory %%{buildroot}/usr/local/bin
install -m 0755 ${PROJECT_SKU} %%{buildroot}/usr/local/bin

install --directory %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/
install -m 644 copyright %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/

install --directory %%{buildroot}/usr/local/share/man/man1/
install -m 644 ${PROJECT_SKU}.1.gz %%{buildroot}/usr/local/share/man/man1/
" >> "${__directory}/SPEC_INSTALL"
```

The content of the `${__directory}/SPEC_FILES` file is the `%files` commands that will be copied over during AutomataCI's automatic spec file generation. You're required to provide the instructions (without the `%files` stanza) in the content assembly function An example would be:

```
        # generate AutomataCI's required RPM spec instructions (FILES)
        printf "\
/usr/local/bin/${PROJECT_SKU}
/usr/local/share/doc/${PROJECT_SKU}/copyright
/usr/local/share/man/man1/${PROJECT_SKU}.1.gz
" >> "${__directory}/SPEC_FILES"
```

Both fragments' specification can be found here:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#an-example-spec-file-for-bello_working-with-spec-files.

5.5.6.6.4 . Collaborating with Automation – Optional Spec Fragment Files

AutomataCI also provides other Spec's Fragment Files for overriding specific fields in the spec file generation such as but not limited to:

1. `${__directory}/SPEC_DESCRIPTION`
2. `${__directory}/SPEC_PREPARE`
3. `${__directory}/SPEC_BUILD`
4. `${__directory}/SPEC_CLEAN`
5. `${__directory}/SPEC_CHANGELOG`

If the `${__directory}/SPEC_DESCRIPTION` spec fragment file is not provided, AutomataCI shall automatically parse and process data from the `${PROJECT_PATH_RESOURCES}/packages/DESCRIPTION.txt` resource file.

If the `${__directory}/SPEC_CHANGELOG` file is not provided ,AutomataCI shall automatically parse and process data from the `${PROJECT_PATH_RESOURCES}/changelog/data/latest` resources data file.

Likewise, stanza (e.g. `%description`) is not required. Only the content is permitted to be in the file.

All specifications are available at:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#an-example-spec-file-for-bello_working-with-spec-files

5.5.6.6.5 . Overriding The Entire Spec File

To override the spec file completely, simply create a fully compilant spec file in the content assembly function at:

`${__directory}/SPECS/${PROJECT_SKU}.spec`

Should AutomataCI detects the existence of such file, the generative function is skipped entirely.

5.5.6.6.6 . Collaborating With Automation – License SDPX Data

RPM requires an explicit declaration of the project's license's SPDX ID. To ensure consistencies across all package ecosystem, AutomataCI automatically source and process the ID from the following file:

```
${__PROJECT_PATH_RESOURCES}/licenses/SPDX.txt
```

Please change the value from there accordingly. Known SPDX IDs are available at: https://spdx.org/licenses/

5.5.6.6.7 . Collaborating With Automation – Project Description Data

AutomataCI requires the long description data for generating the RPM's spec file's `%description` field facilitated by:

```
${__PROJECT_PATH_RESOURCES}/packages/DESCRIPTION.txt
```

Hence, please update the data there for consistencies across all package ecosystems.

5.5.6.6.8 . Distributed Source Code Package

To create a source code package, simply place an empty file with the name complying to the following pattern:

```
{PROJECT_SKU}-src_{OS}-{ARCH}
```

This triggers the package job to recognize it as a target and you can assemble the `BUILD/` path directly. The place-holding file can be created in the Build job recipe phase.

Keep in mind that by doing so, the `$PROJECT_SKU` value used in the automation shall automatically add a "`-src`" suffix for avoiding conflict with the binary package counterpart when both forms are distributed simultaneously.

RPM requires the OS and ARCH to be specific so the "any" ominous value is not available. Hence, please consider which OS and ARCH you wish to facilitate the development in order to restrict the development environment boundaries.

5.5.6.6.9 . Testing Packaged RPM's Health

To test the packaged RPM's health, simply use the following command:

```
$ rpm -K <package>.rpm
<package>.rpm: digests OK
```

If something goes wrong, `rpm` will report out for you.

5.5.6.7 . Red Hat Flatpak (Flatpak)

Red Hat's Flatpak (also known as "Flatpak") is a cross-Linux platform with sandbox capabilities to securely and peacefully distributing applications across the Linux OSes. It is an exclusive distribution dedicated for Linux OSes both Red Hat and Debian alike.

AutomataCI supports Flatpak directly using Flatpak official specification located here: https://docs.flatpak.org/en/latest/introduction.html. To ensure a consistent output metadata across other distribution channels, AutomataCI generates the required `manifest.yml` internally while still permitting developer to overwrite it at the content assembly phase.

Before begin to construct your own job recipe, it is vital to understand at least the specification mentioned earlier alongside the following:

1. https://www.freedesktop.org/software/appstream/docs/chap-Metadata.html
2. https://specifications.freedesktop.org/menu-spec/latest/apa.html

5.5.6.7.1 . Supported Platform

AutomataCI can only build Flatpak output in the following build environment:

UNIX (Debian based)	UNIX (Red Hat based)	MacOS	Windows
Supported	**???**	**Not Supported**	**Not Supported**

5.5.6.7.2 . Content Assembly Function

The content assembly function is:

```
PACKAGE::assemble_flatpak_content() {
     …
}
```

Since both Windows and MacOS do not support `flatpak` due to the Linux kernel requirement, there is no Windows or MacOS counterparts.

In this function, the package job shall pass in the following positional parameters:

```
__target="$1"
__directory="$2"
__target_name="$3"
__target_os="$4"
__target_arch="$5"
```

The `$__directory` variable should point to the workspace directory. The objective is to assemble the "to be installed" file structure into this directory and let `flatpak-builder` to build the flatpak package using a `manifest.yml` file, either generated by AutomataCI or manually overridden.

For example, the given `$__target` variable that is pointing to the currently detected and built binary program is usually being copied as `__directory/${PROJECT_SKU}`.

AutomataCI provides the ability to overrides any existing required files (see below). If these files are absent, AutomataCI shall generate one using its generator functions. Be warned that creating these required files can be a cumbersome effort (due to its steep technical debt). Hence, it is recommended to just focus on constructing the package's data path and leave the rest of the required files to the AutomataCI generative function.

If the function is unused, simply supply a single line with "`return 0`" is suffice to inform the shell that its does nothing.

5.5.6.7.2.1 . Required Files

As specified by Debian engineering specifications, there are 5 known required files:

1. `manifest.{yml||json}`
2. `appdata.xml` ← `${PROJECT_PATH_RESOURCES}/packages/flatpak.xml`
3. `icon.svg` ← `${PROJECT_PATH_RESOURCES}/icons/icon.svg`
4. `icon-48x48.png` ← `${PROJECT_PATH_RESOURCES}/icons/icon-48x48.png`
5. `icon-128x128.png` ← `${PROJECT_PATH_RESOURCES}/icons/icon-128x128.png`

These files follow strict format and content as specified in the Flatpak specification. To ensure consistency is secured and to prevent introducing any unnecessary difficulty, AutomataCI will generate these files when they're not being manually overridden during the content assembly function execution. Due to the complexities involved with Flatpak (e.g. sandbox management), ***it is highly recommended not to override `manifest.yml` OR `manifest.json` file of your choice***.

The `appdata.xml`, `icon.svg`, `icon-48x48.png`, and `icon-128x128.png` are responsible for marketing your Flatpak package in the public repositories. These files have their own respective template and AutoamtaCI would just copy them directly into the workspace directory. The documentations are available at:

1. https://docs.flatpak.org/en/latest/freedesktop-quick-reference.html
2. https://specifications.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html

5.5.6.7.3 . Branch Management

AutomataCI treats each of the Flatpak branches as the supported architecture. Hence, should your project supports multiple architectures by default, your Flatpak repository should checkout the branches in accordance to the CPU type. Read the following for more info:

1. https://docs.flatpak.org/en/latest/using-flatpak.html#identifier-triples

The `default-branch` is set to "any".

5.5.6.7.4 . Sandbox Permission

AutomataCI relies on `${PROJECT_PATH_RESOURCES}/packages/flatpak.perm` to populate the sandbox permission in the manifest file (`finish-args` field). Read more here: https://docs.flatpak.org/en/latest/manifests.html#finishing

This .perm file **must be 1 permission per line as it will be assembled as an array element** by AutomataCI. Hence, you **should only add or remove the required permissions inside the specified `${PROJECT_PATH_RESOURCES}/packages/flatpak.perm` only**. The list of Sandbox permissions are available at: https://docs.flatpak.org/en/latest/sandbox-permissions-reference.html

Please note that there are blacklisted permissions listed in https://docs.flatpak.org/en/latest/sandbox-permissions.html when constructing your list.

5.5.6.7.5 . Directory-based Output

Due to `flatpak-builder`'s output nature, all successful Flatpak packages are directory-based housing the required files and directory structure for `flatpak-builder` to export in the Release job later.

You're free to inspect the output directories but leave them as it is to avoid pipeline breakage later.

5.5.6.7.6 . Screenshots

The screenshots for the `Appdata.xml` file according to the specification here (https://www.freedesktop.org/software/appstream/docs/chap-Metadata.html#tag-screenshots) are best hosted elsewhere and back-linked into the XML data file. There are no signs of the screenshots can be loaded from the package internally.

5.5.6.7.7 . Adding Custom Files

Aside from assembling the custom files via the `PACKAGE::assemble_flatpak_content` function, AutomataCI prepares a collaborative `${PROJECT_PATH_RESOURCES}/packages/flatpak.yml` for one to provide the installation instructions. These instructions shall be appended to the generated `manifest.yml`'s modules fields.

For example, say, a `demo.pdf` document file is made available, then the manifest template YAML file: `${PROJECT_PATH_RESOURCES}/packages/flatpak.yml` should have the an installation instruction like:

```
# ...
modules:
  - name: demo-instruction
    buildsystem: simple
    build-commands:
      - install -D demo.pdf /app/docs/appname-demo.pdf
    sources:
      - type: file
        path: demo.pdf
```

5.5.6.8 . PyPi Library Module (Python)

AutomataCI supports PyPi library module construction through the use of Python 'twine' and 'wheel' libraries. In order to make sure there is a full compliance with Python, both libraries shall be installed using pip command which can be achieved via Prepare job recipe.

The PyPi library module's specifications shall be compliant with the following specifications:

1. https://docs.python.org/3/distutils/setupscript.html
2. https://pypi.org/project/twine/

AutomataCI employs the clean-slate library assembling (similar to first time upload to PyPi) for consistency assurances and for providing maximum freedom to developers in the case of library-app repository use.

AutomataCI relies heavily on $PROJECT_PYTHON environment configuration (set in the repo's CONFIG.toml) file to facilitate PyPi library module construction. Should this configuration is not set (which indicates the repository is not a Python project), this packager shall be ignored entirely.

5.5.6.8.1 . Supported Platform

Should twine is installed as instructed, then PyPi Library Module construction supported platforms are as follow:

UNIX (Debian based)	UNIX (Red Hat based)	MacOS	Windows
Supported	**Supported**	**Supported**	**Supported**

5.5.6.8.2 . Content Assembly Function

The content assembly function for UNIX OS is:

```
PACKAGE::assemble_pypi_content() {
      …
}
```

The content assembly function for Windows OS is:

```
PACKAGE-Assemble-PyPi-Content() {
      …
}
```

In this function, the package job shall pass in the following positional parameters:

```
__target="$1"
__directory="$2"
__target_name="$3"
__target_os="$4"
__target_arch="$5"
```

The `$__directory` variable should point to the workspace directory. The objective is to assemble the Python library "as it is" into this directory and let `twine` and Python to construct the library module. Then finally, script the required `setup.py` file to assemble the library's metadata.

Keep in mind that the given `$__target` variable is usually pointing to a dummy source code target. To check its type, simply use `FS::is_target_a_source` function from FS library and

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process usually for non-compliant technology or not needed
(b) **1** – Error is found
(c) **0** – All good and proceed.

If the function is unused or wanting to disable it, simply supply a single line with "`return 10`" is suffice.

5.5.6.8.3 . Required Files

As specified by the Python distutils library (https://docs.python.org/3/distutils/setupscript.html), the content directory must have the following required files:

1. `setup.py`
2. your library source codes assembled in your end-user's importing manner

Should the `setup.py` file is missing, AutomataCI shall generate a default file on-behalf to fulfill the construction requirement. **Due to its complexities, you are strong encouraged to generate the file during the content assembling function phase to match your actual Project requirement**.

5.5.6.9 . Docker

Docker containerization is a promised cross-platform capable, horizontally scalable, and automated orchestrate-capable container packaging solution to answer massive reliable service needs. As specified by its document (https://docs.docker.com/build/building/base-images/), Docker focus solely on Linux OS only with multiple CPU architecture supports.

AutomataCI supports Docker container packaging support using the following official documentations:

1. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
2. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
3. https://medium.com/@kelseyhightower/optimizing-docker-images-for-static-binaries-b5696e26eb07
4. https://docs.docker.com/build/building/multi-platform/#building-multi-platform-images
5. https://docs.docker.com/build/building/base-images/#create-a-simple-parent-image-using-scratch
6. https://docs.docker.com/engine/reference/builder/

Due to the fact that the product was built via AutomataCI Build job recipe, there is no need to re-build and create a massive-sized container (usually 100+MB~>1GB). Hence, AutomataCI employs Go's approach where the product's `linux-amd64` version is given the highest priority to be purely statically built and then be added into a `scratch` container. This approach generates a very small docker image making Docker's distribution easier.

AutomataCI shall:

1. Build a Docker image stored locally; AND
2. Save a `.tar` version into the `$PROJECT_PATH_PKG` directory that is compatible with `docker load` command.

5.5.6.9.1 . Supported Platform

Currently, Docker image's packaging is supported in the following platforms:

UNIX (Debian based)	UNIX (Red Hat based)	MacOS	Windows
Supported	Supported (Untested)	Not Supported	???

5.5.6.9.2 . Content Assembly Function

The content assembly function for UNIX OS is:

```
PACKAGE::assemble_docker_content() {
      …
}
```

The content assembly function for Windows OS is:

```
PACKAGE-Assemble-DOCKER-Content() {
      …
}
```

In this function, the package job shall pass in the following positional parameters:

```
__target="$1"
__directory="$2"
__target_name="$3"
__target_os="$4"
__target_arch="$5"
```

The `$__directory` variable should point to the workspace directory. The objective is to assemble all the required files and resources into this directory and generate the required `Dockerfile` (named as it is, see later section). For example, the __target is usually copied over and renamed as your project SKU instead (e.g. $PROJECT_SKU in Unix OS).

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process usually for non-compliant technology or not needed.
(b) **1** – Error is found.
(c) **0** – All good and proceed.

If the function is unused or wanting to disable it, simply supply a single line with "`return 10`" is suffice.

5.5.6.9.3 . Required Files

As specified by the Docker, the content directory must have the following required files:

1. **`Dockerfile (Name as it is; no changes allowed)`**
2. your program

Dockerfile can be generated “on-the-fly” in the content assembling function using the `FS::write_file` function. A typical format would be:

```
        __target="${1##*/}"
        FS::write_file "${__directory}/Dockerfile" "\
# Defining baseline image
FROM --platform=${__target_os}/${__target_arch} scratch
MAINTAINER ${PROJECT_CONTACT_NAME} <${PROJECT_CONTACT_EMAIL}>

# Defining environment variables
ENV ARCH ${__target_arch}
ENV OS ${__target_os}
ENV PORT 80

# Assemble the file structure
COPY .blank /tmp/.tmpfile
ADD ${PROJECT_SKU} /app/bin/${PROJECT_SKU}

# Set network port exposures
EXPOSE 80

# Set entry point
ENTRYPOINT [\"/app/bin/${PROJECT_SKU}\"]
"
        if [ $? -ne 0 ]; then
                return 1
        fi
```

Should the Dockerfile is missing, the Package CI job shall fail immediately.

5.5.7 . Release

Release Job is responsible for publishing all the compiled packages to their respective distribution ecosystems. Since these ecosystem distribution processes are usually unchanged, AutomataCI has them built-in for generating the necessary packages output for later Release CI job. This also means that this particular CI job rarely needs a customized job recipe.

AutomataCI detects all the known packages in the `PROJECT_PATH_ROOT/PROJECT_PATH_PKG` (defined in `CONFIG.toml`) with their respective right tools. Right before any execution, it shall detects all the associated technologies' pre-processor functions and run them. Then, it shall loop through all known packages and process them using its internal functions. Once done, it shall runs all the associated technologies' pre-processor functions. Upon completion, the content within `PROJECT_PATH_ROOT/PROJECT_PATH_PKG` directory is ready for any remaining manual upload (e.g. GitHub Release).

5.5.7.1 . Cryptography Requirements

Do note that some ecosystems require cryptography implementations such as but not limited to GPG signing for `.deb` and `.rpm` package types. To protect the cryptography private keys from being exposed out (via 3rd-party service providers' contractors intentionally or unintentionally), **it is always remain as secrets to your side by operating locally**.

5.5.7.2 . Special Custom Implementations

Unlike all other jobs, Package Job recipe **requires a compulsory custom CI job recipe** to supply the required package content assembly functions. Each deployed technologies shall support its own pre-processor and post-processor function via its `release_unix-any.sh` and `release_windows-any.ps1` in their `PROJECT_PATH_CI` directory. The function should have their technology names in it like, for Python:

```
RELEASE::run_python_post_processor() { … }
RELEASE::run_python_pre_processor() { … }
```

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the release process entirely (only makes sense in pre-processor).
(b) **1** – Error is found.
(c) **0** – All good and proceed.

Usually, `pre-processor` function is for making some tidy-up works before the actual release job and `post-processor` function is for facilitating any left-over or custom work to be done. Each technology has its turn to operate its functions as long as it's enabled in the project.

5.5.7.3 . Local Static Hosting Repository

By default, AutomataCI deploys local static hosting repository (e.g. using GitHub Wiki in the repo) to publish certain types of packages (namely `.deb`, `.rpm`, and `.flatpak`) in an ordered manner so that the end-user can source directly.

5.5.7.4 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}/

It processes its output in the following directories:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_RELEASE}/

5.5.7.5 . Archive (.tar.xz | .zip)

5.5.7.6 . Debian Package (.deb)

AutomataCI uses Reprepro external technology to process the `.deb` Debian package into an APT friendly repository for customers to deploy using the famous "`apt get install`" or "`apt install`" command.

Due to the requirement of Reprepro, GPG cryptography signature is required for the publications.

The destination is set to:

`${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/deb`

where: `${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}` is the `PROJECT_STATIC_REPO` directory.

The original package file is left in-tact in case some customer wants to install manually.

5.5.7.7 . Red Hat Package (.rpm)

5.5.7.8 . Red Hat Flatpak (Flatpak)

5.5.7.9 . PyPi Library Module (Python)

5.5.7.10 . Docker

5.5.8 . Compose

Compose Job is responsible for generating the project's documentations (e.g. website, PDFs and etc) artifacts for Publish job.

TODO – under construction.

5.5.9 . Publish

Publish Job is to publish the composed documentations to the corresponding publication ecosystem.

TODO – under construction.

5.5.10 . Clean

Clean Job is to remove all operating artifacts except the PROJECT_PATH_TOOLS directory for a clean operation.

TODO – under construction.

5.5.11 . Purge

Purge Job is to remove everything including PROJECT_PATH_TOOLS directory and restore the project to its initial state.

TODO – under construction.

6 . Epilogue

That’s all from us. We wish you would enjoy the project development experiences to your delights.