# AutomataCI Engineering Specification

Version 1.7.0 International English (en), November 2023

# Table of Contents

# 1 . Prologue

First of all, thank you for selecting and using my AutomataCI solution. This document is a developer-specific specification for deploying and maintaining AutomataCI in your source codes repository. In case of any inquiry, please feel free to contact me or my team at:

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

# 2 . Customers' Quick-Start

This section covers a quick re-cap for the experienced customers deploying AutomataCI without requiring to go through the entire specification.

For newcomers, please do go through the specifications at least once in order to understand how AutomataCI operates and manages the project's repository.

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 . [OPTIONAL] Provide SECRETS.toml

Depending on whether you have access to your static release repository, you may need to initiate the following environment variables for your project based on needs. AutomataCI recommends setting them in a *SECRETS.toml* file. There is a template file made available inside *automataCI/* directory (called *SECRETS-template.toml*). Please duplicate and then rename it before use.

**IMPORTANT REMINDER**: The TOML can only and strictly accept "***key = 'value'***" format ONLY. Don't get too fancy with other TOML formats.

| **Variables** | **Descriptions** | **Example Value** |
|---|---|---|
| *TERM* | Configure terminal output settings | xterm-256color |
| *CONTAINER_USERNAME* | Your container registry username | username_in_string |
| *CONTAINER_PASSWORD* | Your container registry password | password_in_string |
| *PROJECT_SIMULATE_RELEASE_REPO* | Set AutomataCI to simulate release repository instead of sourcing directly. Leave this unset or empty if you want to interact directly with release repo. | true |
| *TWINE_USERNAME* | Your Python PyPi Account username provided by the package server for your Python package publication. If API token is used, your username is likely a "*__token__*" string. | Username<br><br>OR<br><br>"__token__" |
| *TWINE_PASSWORD* | Your Python PyPi Account's password or API token for your Python package publication. | secret_in_string |
| *CARGO_PASSWORD* | Your Cargo registry password or API token for Rust Cargo's crate publication. | secret_in_string |

## 2.4 . To Check and Setup Host Machine Environment

To quickly configure the host machine for specific tech requirements, simply:

```
$ ./ci.cmd env
```

Upon completion, your host machine should be ready for the setup CI job.

## 2.5 . To Setup The Repository for Development

To quickly setup the project repository simply:

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

Upon completion, your repository is ready for development.

## 2.6 . To Start A Development

To start the development after setting up, simply:

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

Upon completion, your terminal should be ready for development.

## 2.7 . 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.

## 2.8 . To Materialize for Host Machine Local Usage

To materialize the product just for the host machine usage, simply:

```
$ ./ci.cmd materialize
```

Upon completion, please check your *{PROJECT_PATH_ROOT}/{PROJECT_PATH_BIN}* for the executable and *{PROJECT_PATH_ROOT}/{PROJECT_PATH_LIB}* for any library.

## 2.9 . 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.

## 2.10 . To Notarize all the Built Products

To notarize all the production-ready product, simply:

```
$ ./ci.cmd notarize
```

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

**GENTLE REMINDER**: it's your duty to have all the secret keys and certificates in-place before executing this command.

## 2.11 . 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.

## 2.12 . 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.

## 2.13 . To Deploy The New Release

To deploy the new release, simply execute the following locally:

```
$ ./ci.cmd deploy
```

Upon completion, you may check the ecosystem that you deployed to.

## 2.14 . To Clean The Repo

To clean the job cycles, simply execute the following locally:

```
$ ./ci.cmd clean
```

Upon completion, you can be rest assured the repository is cleansed as per you specifications.

## 2.15 . To Purge The Repo

To purge the repo entirely to its initial state, simply execute the following locally:

```
$ ./ci.cmd purge
```

Upon completion, your repo is back to its original state where you need to setup your tooling and everything from scratch again.

## 2.16 . To Customize CI Jobs

Only alters the tech-specific CI job script. Example:

Baseline – *${PROJECT_PATH_ROOT}/${PROJECT_PATH_SOURCE}/.ci/[job]-unix-any.sh*
Python – *${PROJECT_PATH_ROOT}/${PROJECT_PYTHON}/.ci/[job]-unix-any.sh*
Go – *${PROJECT_PATH_ROOT}/${PROJECT_GO}/.ci/[job]-unix-any.sh*

**REMINDER**: **DO NOT edit anything inside *automataCI/* directory**. All changes in that directory shall be overridden during AutomataCI upgrade (see below).

## 2.17 . To Upgrade AutomataCI

This is a multi-steps checklist to make sure the AutomataCI is safely upgraded:

1. **Determine the business need** – check the release notes and make sure there is something you need in the latest version. Otherwise, you don't have to upgrade towards latest and greatest at all.
2. **Overwrite the *automataCI/* directory** – Download the later version of AutomataCI and only overwrites the local *automataCI/* directory.
3. **Cross-check *CONFIG.toml*** – Cross-check the later version's changes and patch it into your project.
4. **Apply additional changes if instructed** – Should the release note instructs additional changes, please cross-check and apply them.
5. **Update the common source's CI job recipes** – First update the common source (e.g. *src/* directory)'s CI job recipe to match the latest version's features and changes.
6. **Update the project's tech-specific CI job recipes** – Update all the project's tech specific CI job recipes based on the later version's features and changes if needed.
7. **Test and re-run** – Test and re-run. Repeat previous step if needed.
8. **Commit the changes** – Should the pipeline is working fine. You can commit and AutomataCI is updated.

# 3 . Engineering Specification

This section is about the specification of the AutomataCI stewarding the source codes repository semi-automatically for fulfilling the long-term sustainable maintenance and development roles.

## 3.1 . Why Another Continuous Integration Solution

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

## 3.2 . 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 anytime. The project is designed for both remote executions in the cloud or even locally. It facilitates heavier resistances and resilience to market changes without hampering the product development and production progress.

Unlike other CI models, **AutomataCI favors the "semi-automatic" approach** where the automation **can also be manually executed stage-by-stage or fully automated**. This provides decent protection against ill-intent vendors from legally extorting or ransoming you via vendor locked-in and after-the-fact business changes. If a CI service provider turns the relationship sour, one can easily switch to another. Another good reason is the decoupling effect done to the CI, allowing developers to specifically test a pipeline job manually and locally.

## 3.3 . 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 knowledge:

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 Microsoft Windows' Batch scripting, you do not need Batch script beyond that scope. If you're still using one, you're doing things the wrong way at least in the AutomataCI perceptive. (Besides, 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.

AutomataCI employs Homebrew (https://brew.sh/) and Chocolatey (https://chocolatey.org/) to manage and supplement most of the foundational software supplies across various OSes (MacOS and Linux are using Homebrew; Windows is using Chocolatey). These managers provides a copy of the procured software locally so in case of any unfortunate Geo-political induced embargo happens to you, your business will not be completely stopped and halted.

### 3.3.1 . Isolating External Infrastructure Services

To counter the external CI service providers' threatening changes, **their service' interfaces is strictly ONLY to call AutomataCI job executions just like a regular developer does in the local machine**. Therefore, most of the time, you will be using AutomataCI's service libraries in both of your POSIX shell and PowerShell scripting instead. By deploying adapter approach, it's easier to maintain and isolate all 3rd-party service providers without hampering the customers' project or AutomataCI maintenance teams.

Due to the fact that CI is an important life-support system for your project, **you're strongly advised not to use any vendor-specific API or functionalities. Anything AutomataCI can't do locally signaling that it is vendor locked-in solution**. 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.

## 3.4 . System Naming Convention

AutomataCI uses the lowercase-ed uname list (see: https://en.wikipedia.org/wiki/Uname#Examples) as its source complying to the following format where they can be interpreted interchangeably:

**{PLATFORM}** = **{OS}-{ARCH}**

Example, for identifying the Host platform system, the environment variables *PROJECT_PLATFORM*, *PROJECT_OS*, and *PROJECT_ARCH* are defined before executing any CI job.

A special value '**any**' is allowed for both OS and ARCH fields denoting that the context is can operate independent of operating system or cpu architecture respectively. Here are some examples:

| **Platform** | **Description** | **Examples Use Cases** |
|---|---|---|
| linux-amd64 | Linux OS + AMD64 CPU only | Kernel binary, .o, .d, etc. |
| linux-any | Linus OS only | Bash script |
| any-any | Any OS & Any CPU | Polygot script |
| darwin-amd64 | Mac OSX + Intel CPU | App binary |
| darwin-arm64 | Mac OSX + M-Series CPU | App binary |
| windows-amd64 | Windows + AMD64 CPU only | Program.exe binary |

### 3.4.1 . Specific Redefinition

It's duly noted that AutomataCI employs these custom values that is being used widely across the software industries:

| ***ARCH*** | ***Description*** | ***Denoted As*** |
|---|---|---|
| i686-64 | Intel Itanium CPU | ia64 |
| i386, i486, i586, i686 | Intel X86 32-bit CPU | i386 |
| x86_64 | Intel/AMD X86 64-bit CPU | amd64 |
| sun4u | TI Ultrasparc | sparc |
| power macintosh | IBM PowerPC | powerpc |
| ip* | MIPS CPU | mips |

| ***OS*** | ***Description*** | ***Denoted As*** |
|---|---|---|
| windows*, ms-dos* | Microsoft Windows OSes | windows |
| cygwin*, mingw*, mingw32*, msys* | Linux Emulator in Windows OSes | windows |
| *freebsd | FreeBSD OSes | freebsd |
| dragonfly* | Dragonfly OSes | dragonfly |
| standalone, unknown, none, any | Bare-metal (e.g. microcontroller images / fat binary) | none |

## 3.5 . Filesystem

The AutomataCI requires at least the following important elements to operate properly:

1. **ci.cmd** – The Polygot script unifying all OSes start point meant for you to trigger a CI job.
2. **automataCI libraries** – A directory housing all the AutomataCI job recipes and function services.
3. **CONFIG.toml** – A simple TOML formatted configuration file that provides the repository's critical parameters for AutomataCI to operate and manage with. It specifies the critical directories of the filesystem alongside their respective explanations. You should go through over there for avoiding duplication here.

### 3.5.1 . Secrets Parameters Configuration File

The **SECRETS.toml** repo config file is a especially prepared optional file that behaves similarly as **CONFIG.toml file but set to be .gitignored** by default. This special config file is meant for supplying environmental parameters that has high confidentiality.

You're free to provide those secret parameters either through this file or define those environment variables directly on your own.

### 3.5.2 . 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.

### 3.5.3 . Repository File Structures

By default, the following file structures are defined:

| | |
|---|---|
| automataCI/ | ➞ house the projects' CI automation scripts. |
| automataCI/ci.sh | ➞ execute AutomataCI in UNIX OS. |
| automataCI/ci.ps1 | ➞ execute AutomataCI in Windows OS. |
| automataCI/services | ➞ house tested and pre-built CI automation functions. |
| bin/ | ➞ default materialized output directory for host's executable files. |
| build/ | ➞ default build output directory. |
| lib/ | ➞ default materialized output directory for host's library files. |
| pkg/ | ➞ default package output directory. |
| src/ | ➞ house common assets and materials (baseline directory). |
| src/packages | ➞ housing all packages control template files. |
| src/icons | ➞ housing all graphics and icon files. |
| src/licenses | ➞ housing all project licensing generative documents. |
| src/docs | ➞ housing all project's document generators. |
| src/changelog | ➞ housing all project's changelog entries data. |
| src/.ci/ | ➞ house common CI job recipes (applies to all tech). |
| src[TECH]/ | ➞ house tech-specific source codes (tech-specific directory). |
| src[TECH]/.ci/ | ➞ house tech-specific source codes CI job recipes (applies to selected tech). |
| tools/ | ➞ default tooling (e.g. programming language's bin/*) directory. |
| tmp/ | ➞ default temporary workspace directory. |
| CONFIG.toml | ➞ configure project's settings data for AutomataCI. |
| SECRETS.toml | ➞ provides secrets related data (ignored by .gitignore). |
| ci.cmd | ➞ CI start point calling *automataCI/ci.sh* or *automataCI/ci.ps1*. |
| .git | ➞ Git version control configuration directory. |

### 3.5.4 . Filename Naming Convention

AutomataCI uses:

1. underscore (_) for context switching; AND
2. dash (-) for separating different subjects for the same context; AND
3. No space is allowed.

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

**{PURPOSE}_{SYSTEM}.{EXTENSION}**

OR

**{PURPOSE}_{OS}-{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.

## 3.6 . CI Jobs

This section covers all the CI jobs' specification. AutomataCI employs a linear story-line style of job executions. In each job, the developer can deploy concurrent or parallel executions as long as the host platform and sanity permits. The story-line itself is serially executed for high clarity and accuracy over speed.

The CI jobs can be customized without modifying the main recipe files via the enabled technologies' **.ci/{purpose}_{OS}-any.{EXTENSION}** job recipe file. For example: when **{PROJECT_PYTHON}** is defined (means the project is deploying Python technology), AutomataCI do seek out the CI job recipe files and execute accordingly in: **{PROJECT_PATH_ROOT}/{PROJECT_PYTHON}/.ci/{purpose}_{OS}-any.{EXTENSION}**

The overall execution flow and sequences are as follow:

1. user only triggers the CI jobs via the **ci.cmd** Polygot script; AND
2. Then the **ci.cmd** sorts out the platform specific data and shall calls by source ("*$ . file*")the CI job recipes housed in the level-1 of **automataCI** directory; AND
3. The corresponding CI job recipes detect both baselines and customized job recipe and execute accordingly unless otherwise specified.

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

1. User executed "**./ci.cmd setup**" command in *Windows* OS with *amd64* CPU, which then executes ***automataCI/ci.ps1*** file.
2. ***automataCI/ci.ps1*** executes **automataCI/common_windows-any.ps1** file.
3. **automataCI/common_windows-any.ps1** detected ***src/.ci/setup_windows-any.ps1*** baseline job and ***srcPYTHON/.ci/setup_windows-any.ps1*** custom job.
4. **automataCI/common_windows-any.ps1** execute the ***srcPYTHON/.ci/setup_windows-any.ps1*** tech-specific job first.
5. **automataCI/common_windows-any.ps1** then execute the ***src/.ci/setup_windows-any.ps1*** baseline job last.

### 3.6.1 . Environment

Environment operates by setting up the host platform OS to facilitate fundamental technologies like programming languages, packagers, OS-related configurations, and etc uniformly using compatible 3rd-party software like Homebrew and Chocolatey.

This is entirely done based on the technologies set in the *CONFIG.toml* configuration file. Example, should **{PROJECT_PYTHON}** is defined, Homebrew and Chocolatey shall install the Python programming language in the host machine accordingly without utilizing root/admin privilege.

The job does not have any customization CI job recipe to deal with. It reads all its configurations via AutomataCI's runtime definitions.

#### *3.6.1.1 . Only Fundamental Technologies*

Although CI job customization are available, it is important to keep it under the technologies' software package manager (e.g. *pip* for Python; *go get* for Go) job to do so.

Different host machine approaches setup differently. Hence, you only use this job when it's absolute necessary.

#### *3.6.1.2 . Operating Parameters*

Depending on OS, this job shall output its installations as defined by HomeBrew and Chocolatey specifications listed below:

1. **Homebrew** – https://docs.brew.sh/Installation
2. **Chocolatey** – https://docs.chocolatey.org/en-us/default-chocolatey-install-reasoning

#### *3.6.1.3 . Angular Framework*

When **PROJECT_ANGULAR** is set, AutomataCI setup 2 set of technologies for operating Angular Framework (https://angular.io/):

1. Setup the required NodeJS (https://nodejs.org/) Javascript runtime engine; AND
2. Angular CLI (https://angular.io/cli).

The conventional commands are as follow:

```
# UNIX System (Linux & MacOS)
$ brew install node
$ npm install -g @angular/cli

# windows
$ choco install node -y
$ npm install -g @angular/cli
```

Should AutomataCI detects any existing installation (e.g. both *npm* and *ng* programs), it shall automatically be skipped prioritizing using the existing ones.

#### *3.6.1.4 . C Programming Language*

Setting up a C Programming Language environment development environment can be very complicated depending on your target device, your host machine's operating system and your host machine's CPU architecture.

While AutomataCI attempts to cover as much build possibilities as possible but it cannot cover 100% of all edge cases. What AutomataCI covers would be the known common microprocessor products (e.g. laptop, PC, server, etc) like MacBook, Linux OS, and Windows' PC + Server uses only.

For embedded use (edge case), AutomataCI is designed to facilitate one but it is the customer's duty to setup the compiler in the host machine. Known edge cases setup are documented in the following sub-sections.

##### 3.6.1.4.1 . From LINUX-AMD64 for AVR Microcontrollers

On *linux-amd64* host building for *avr* Microchip's (ATMEGA series) microcontrollers, you need to setup the avr toolkit accordingly. Example on Debian-based OS:

```
$ apt install avr-libc avrdude binutils-avr gcc-avr srecord -y
$ apt install gdb-avr simulavr -y
```

##### 3.6.1.4.2 . From DARWIN-AMD64/ARM64 for AVR Microcontrollers

On MacOS (amd64 CPU or arm64 "M Series"), you may want to install external cross-compilers such but not limited to:

```
$ brew tap osx-cross/avr
$ brew install avr-gcc
```

Reference: https://github.com/osx-cross/homebrew-avr#installing-homebrew-avr-formulae

#### 3.6.1.4.3 . From LINUX-AMD64 for Known Microprocessors

On *linux-amd64* host building for other supported microprocessor systems, you need to setup the cross-compilers accordingly. Example on Debian-based OS:

```
$ apt install build-essential crossbuild-essential-amd64 \
        crossbuild-essential-arm64 \
        crossbuild-essential-armel \
        crossbuild-essential-armhf \
        crossbuild-essential-i386 \
        crossbuild-essential-mips \
        crossbuild-essential-mips64 \
        crossbuild-essential-mips64el \
        crossbuild-essential-mips64r6 \
        crossbuild-essential-mips64r6el \
        crossbuild-essential-mipsel \
        crossbuild-essential-mipsr6 \
        crossbuild-essential-mipsr6el \
        crossbuild-essential-powerpc \
        crossbuild-essential-ppc64el \
        crossbuild-essential-s390x \
        -y
```

Should root access for setting up development environment is unavailable, brew installing the following cross-compilers work as a redundancy (NOTE: already facilitated by AutomataCI's ENV job) should any of the required native (cross-)compilers is unavailable):

```
$ brew install \
        aarch64-elf-gcc \
        arm-none-eabi-gcc \
        riscv64-elf-gcc \
        x86_64-elf-gcc \
        i686-elf-gcc \
        mingw-w64 \
        llvm \
        emscripten \
        gcc
```

On Windows host machine, when Chocolatey (https://community.chocolatey.org/) is used, the following packages are advised to be setup in the host machine (NOTE: not all packages can be installed automatically by AutomataCI's ENV job):

```
$ choco install gcc-arm-embedded -y
$ choco install mingw -y
$ choco install emscripten -y  # NOTE: this has bug causing automation to fail.
```

For C Programming Language on Windows, due to the extreme complexity in Windows OS, AutomataCI does not support C/C++ cross-compilation capable development on Windows' host machine. It's better to setup a Linux VM and develop in there. Also, AutomataCI does not support proprietary C/C++ development (e.g. C# and VS related products). They are likely comes with their own IDE system and AutomataCI encourages you to use them instead.

#### *3.6.1.5 . Nim Programming Language*

When **PROJECT_NIM** is enabled for Nim Programming Language, AutomataCI shall setup the entire C Programming Language environment setup stated previously alongside its own as the default compiler:

```
# UNIX System (Linux & MacOS)
$ brew install nim

# windows
$ choco install nim -y
```

Although Nim is very flexible when generating a compiled source code (not to confused with executable), AutomataCI prioritizes C for its simplicity sake.

##### 3.6.1.5.1 . Nimble Packages

There is a special requirement where nimble only allows:

(a) 1 *{main}.nim* source code; AND
(b) 1 *{main}pkg* directory inside the source directory (default is *srcNIM*).

See: https://github.com/nim-lang/nimble#creating-packages for more info. The *{main}* naming keyword MUST match your *$PROJECT_SKU* value. Otherwise, it won't work. [Reference: https://github.com/nim-lang/nimble/blob/master/nimble.nimble]

AutomataCI generates the *.nimble* configuration file per build due to *Nimble*'s limitation of supporting cross-compilation (see: https://github.com/nim-lang/nimble/issues/510).

#### *3.6.1.6 . Rust Programming Language*

Due to the restriction imposed by Rust Programming Language that requires the compiler to be installed only via *rustup* program (see: https://rust-lang.github.io/rustup/installation/index.html and https://rust-lang.github.io/rustup/installation/windows.html), AutomataCI shall not perform any Homebrew or Chocolatey installation for Rust.

The setup roles and tasks are completely taken over by Setup CI job.

### 3.6.2 . Setup

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

Example, for Python programming language, it setups a Python Virtual Environment (*venv*) directory usually located in the **${PROJECT_PATH_TOOLS}/python-engine** directory.

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/setup_unix-any.sh* via *PROJECT_PYTHON*); AND THEN
2. baseline job recipe (e.g. *src/.ci/setup_unix-any.sh* via *PROJECT_PATH_SOURCE*)

To activate the virtual environment for development use, see “Start” CI Job (next section) that will print the technology-specific’s startup instruction.

#### *3.6.2.1 . Go Programming Language*

It's notable that although not required, AutomataCI will still setup and create a virtual environment for Go Programming Language to group the project-specific installations and libraries under its engine. This prevents the Go engine to clog the development machine by installing stuffs all over the places creating a false-sense of supports.

This job place it output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/go-engine

#### *3.6.2.2 . Nim Programming Language*

It's notable that although not required, AutomataCI will still setup and create a virtual environment for Nim Programming Language to group the project-specific installations and libraries under its engine. This prevents the Nim engine to clog the development machine by installing stuffs all over the places creating a false-sense of supports.

This job place it output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/nim-engine

#### *3.6.2.3 . Python Programming Language*

AutomataCI will still setup and create a Python virtual environment to group the project-specific installations and libraries under its engine.

This job place its output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/python-engine

#### *3.6.2.4 . Rust Programming Language*

AutomataCI uses a local copy of *rustup.sh* shell script and sourcing *rustup.exe* script for download safety. It setup not only the localized environment but also Rust compilers and its tools.

This job place its output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/rust-engine

### 3.6.3 . Start

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

Example, for Python programming language, it provides the instruction where to activate the Python Virtual Environment (*venv*).

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/start_unix-any.sh* via *PROJECT_PYTHON)*; AND THEN
2. baseline job recipe (e.g. *src/.ci/start_unix-any.sh* via *PROJECT_PATH_SOURCE*)

#### *3.6.3.1 . 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]

### 3.6.4 . 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.

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/prepare_unix-any.sh* via *PROJECT_PYTHON)*; AND THEN
2. baseline job recipe (e.g. *src/.ci/prepare_unix-any.sh* via *PROJECT_PATH_SOURCE*)

#### *3.6.4.1 . Operating Parameters*

This job takes the following as its output:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/
{PROJECT_PATH_ROOT}/{PROJECT_PATH_TEMP}/

#### *3.6.4.2 . Angular Framework*

For Angular framework, when using the provided *srcANGULAR/* directory, it's executing the npm packages installation command:

```
$ npm install
```

#### *3.6.4.3 . Go Programming Language*

For Go programming language, when using the provided *srcGO/* directory, it's executing the "*go get*" command and source all the dependencies listed in the *go.mod* file. The similar manual command is as follows:

```
$ cd srcGO && go get . &&  cd ..
```

#### *3.6.4.4 . Nim Programming Language*

For Nim programming language, although *nimble* is setup properly but does not automatically install all dependencies from the *.nimble* file, it's required for the project author to explicitly instruct the installation. Example:

```
$ nimble install <package>
```

#### *3.6.4.5 . Python Programming Language*

For Python programming language, when using the provided *srcPYTHON/* directory, it's executing the pip packages installation command:

```
$ pip install -r requirements.txt
```

#### *3.6.4.6 . Rust Programming Language*

For Rust programming language, when using the provided *srcRUST/* directory, it's executing the cargo packages fetching command:

```
$ cargo fetch
```

### 3.6.5 . 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.

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/test_unix-any.sh* via *PROJECT_PYTHON)*; AND THEN
2. baseline job recipe (e.g. *src/.ci/test_unix-any.sh* via *PROJECT_PATH_SOURCE*)

#### *3.6.5.1 . 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}/

#### *3.6.5.2 . Angular Framework*

The provided *srcANGULAR* CI job do check for **$PROJECT_SIMULATE_RELEASE_REPO** environment variable before executing *ng test* command. This is mainly due to Karma test framework depending on a Blink-based browser to execute all its unit test suites.

Unlike guided by most Internet articles, AutomataCI discourages using *PhantomJS* to operate all the unit test codes mainly due to Angular's frontend art-directive nature where it depends on human artistic acceptances to realistically judge an art. Hence, AutomataCI shall simulate the test run instead.

##### 3.6.5.2.1 . Angular's Native Karma + Jasmine Test Framework

AutomataCI uses the Angular's native Karma and Jasmine test framework to operate Angular test components. Hence, all existing knowledge and test suites can be directly reusable without additional learning.

Reference: https://angular.io/guide/testing

#### *3.6.5.3 . C Programming Language*

AutomataCI offers its very own unit-testing facility for C Programming Language where it is missing from its native compilation ecosystem. AutomataCI challenges the fundamental of unit testing paradigm where each tech suite are statically compiled executable and execute separately. This is done for the provided *srcC/* sample library, where the *greeter_any-any.c* library function is being tested with its own unit test codes (*greeter_any-any_test.c*).

##### 3.6.5.3.1 . Self-Contained Statically Compiled Executable

AutomataCI scans for all *_test.c* suffix source codes across the *${PROJECT_PATH_ROOT}/${PROJECT_C}* directory recursively. **Each of these test file are self-contained main file** which then be compiled into its own executable based on the host machine.

Once the compilation stage is completed, AutomataCI then execute each of these executable. Should any of them fails, the test shall be concluded as failing entirely.

The build directory can be manually verified and analyzed in the *${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/ctest_** directory.

Note that due to the way AutomataCI is designed, you should avoid testing cross-compilation build sequences in this Test CI job. Instead, develop the Build CI job properly and let Build CI reports those errors instead. The focus is on testing your C codes.

#### *3.6.5.4 . Go Programming Language*

The provided $PROJECT_GO (default: *srcGO/*) CI job uses its go test and go tool features to execute all visible test suites and its test cases, profile the test coverage scope, and lastly generates a test coverage heatmap HTML file for pinpoint testing.

The generated report is located in:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/go-test-report/

#### *3.6.5.5 . Nim Programming Language*

Likewise with C Programming Language, there were no successful run done in Nim using their recommended Testament package (https://nim-lang.org/docs/testament.html). Hence, AutomataCI fallback to using its own unit-testing facility. It challenges the fundamental of unit testing paradigm where each tech suite are statically compiled executable and execute separately. This is done for the provided *srcNIM/* sample library, where the *Greeter_any-any.nim* library function is being tested against its own unit test codes (*Greeter_any-any_test.nim*).

##### 3.6.5.5.1 . Self-Contained Statically Compiled Executable

AutomataCI scans for all *_test.nim* suffix source codes across the *${PROJECT_PATH_ROOT}/${PROJECT_NIM}* directory recursively. **Each of these test file are self-contained main file** which then be compiled into its own executable based on the host machine.

Once the compilation stage is completed, AutomataCI then execute each of these executable. Should any of them fails, the test shall be concluded as failing entirely.

The build directory can be manually verified and analyzed in the *${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/nim-test_** directory.

Note that due to the way AutomataCI is designed, you should avoid testing cross-compilation build sequences in this Test CI job. Instead, develop the Build CI job properly and let Build CI reports those errors instead. The focus is on testing your C codes.

#### *3.6.5.6 . Python Programming Language*

The provided srcPYTHON CI job uses its *unittest* pip package alongside *coverage* pip package for generating the source code test coverage heatmap HTML file.

The generated report is located in:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/go-python-report/

##### 3.6.5.6.1 . Dependencies Installations

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!
pdoc3
coverage
pyinstaller
wheel
twine
```

#### *3.6.5.7 . Rust Programming Language*

The provided **$PROJECT_RUST** (default *srcRUST/* directory) CI job uses *grcov* and its *llvm-tools-preview* component installed via *cargo* and *rustup* respectively. These dependencies are currently installed by default during the initial rust engine setup in the Setup CI Job. Although the facilities provided by Rust currently is quite messy, it is still capable of generating code coverage heatmap for developer to perform pinpoint accurate testing while executing the unit testing facilities.

AutomataCI executes this Test CI Job for Rust Programming Language based on the following documentations:

1. https://doc.rust-lang.org/rustc/instrument-coverage.html
2. https://github.com/mozilla/grcov
3. https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html

By using the provided *srcRUST/* workspace, AutomataCI shall execute Rust's native unit testing facilities and generate the code coverage heatmap report.

The generated report is located in:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/rust-test-report/

### 3.6.6 . Materialize

**Materialize job is the same as Build Job (see later section) with only 1 difference: *it only builds for the host machine to utilize the product locally***. Materialize job compiles to the following directory instead (which is different from Build job):

${PROJECT_PATH_ROOT}/${PROJECT_PATH_BIN}

This job is created to facilitate efficient and easy to build for

1. Source-based distribution where cross-compilation is not required such as but not limited to *Homebrew* and *Chocolatey* ecosystems; AND
2. Manual git setup and unpack implementations.

Depending on project technologies (e.g. C Programming Language), this job ensures the end-user can reproduce and utilize the AutomataCI built-in tools and services with 100% consistency and without needing to develop a workaround. It's duly noted that every build specifications and differences shall be the same as Build job (see later section).

#### *3.6.6.1 . 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_BIN}/
{PROJECT_PATH_ROOT}/{PROJECT_PATH_DOCS}/

### 3.6.7 . 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, latest changelog entries, and compose documentations.

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 product's 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.

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}[{EXTENSION}]

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

To generate a source package for supporting linux-amd64 development, a placeholder file is created (either by touching):

1. myproc-src_linux-amd64

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/build_unix-any.sh* via *PROJECT_PYTHON)*; AND THEN
2. baseline job recipe (e.g. *src/.ci/build_unix-any.sh* via *PROJECT_PATH_SOURCE*)

#### *3.6.7.1 . Job Recipe Customization*

To customize the job, the modifications shall be done to the tech-specific Build CI job recipe. Example:

For Python, they are:

{PROJECT_PATH_ROOT}/{PROJECT_PYTHON}/{PROJECT_PATH_CI}/build_unix-any.sh
{PROJECT_PATH_ROOT}/{PROJECT_PYTHON}/{PROJECT_PATH_CI}/build_windows-any.ps1

#### *3.6.7.2 . 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}/
{PROJECT_PATH_ROOT}/{PROJECT_PATH_DOCS}/

#### *3.6.7.3 . Angular Framework*

AutomataCI run *ng build* that produces the static artifact output files specified in the *$PROJECT_ANGULAR/angular.json* file as follows:

```
...
    "architect": {
     "build": {
       "builder": "@angular-devkit/build-angular:browser",
       "options": {
         "outputPath": "../public",
         "index": "src/index.html",
         "main": "src/main.ts",
         "polyfills": [
           "zone.js"
         ],
...
```

By default, to ensure maximum compatibility, the *angular.json* is set to dump the output to: *${PROJECT_PATH_ROOT}/${PROJECT_PATH_DOCS}* directory (defined by *CONFIG.toml* file). Anything other controls are solely under Angular CLI on its own.

##### 3.6.7.3.1 . Documentation Output Directory

Unlike other technologies, being a front-end framework, ***AutomataCI configures Angular to dump its output into $PROJECT_PATH_DOCS instead of $PORJECT_PATH_BUILD directory***. As recommended by Angular, the artifacts are the one to be re-distributed instead of the primary repository.

#### *3.6.7.4 . Chocolatey Ecosystem*

Chocolatey (https://docs.chocolatey.org/) ecosystem is a simple ecosystem (some metadata included in a zip file). After careful analysis with all its developer's use cases permutations, the only sensible and sustainable way to proceed is packing its own archive package (in *.zip* format) containing the required files for it.

Hence, in order to support and enable Chocolatey packaging, a placeholder file shall be created (either by touching or write an empty file) for Package Job to pick up the task later:

*myproc-chocolatey_any-any*

### *3.6.7.5 . C Programming Language*

For building a product using C Programming Language, AutomataCI is packed with all the built infrastructure natively without requiring additional build tools installation that can add additional knowledge taxes like learning *Makefile, make*, *cmake*, ... etc. AutomataCI supports building the C executables and importable libraries with cross-platform capabilities by default. These implementations are designed based on the following specifications:

1. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
2. https://wiki.debian.org/Hardening
3. https://ntrs.nasa.gov/api/citations/19950022400/downloads/19950022400.pdf
4. https://www.kernel.org/doc/html/v4.10/process/coding-style.html
5. https://www.kernel.org/doc/html/v4.10/admin-guide/index.html
6. A book "Advanced Linux Programming" by Mark Mitchell, Jeffrey Oldham, Alex Samuel

AutomataCI supplies a *BUILD::compile* (or in Windows: *BUILD-Compile*) function to replace the existing 3rd-party dependencies' *make* and *Makefile* implementations. This function is defined as follows:

```
BUILD::compile \
        "binary"  \             # Param 1 – define the output type ('binary' or 'library')
        "linux" \               # Param 2 – define the target OS.
        "amd64" \               # Param 3 – define the target ARCH.
        "main-bin.txt" \        # Param 4 – define the relative path to the build list text file.
        "-Wall -Wextra ..." \   # Param 5 – compiler arguments / settings.
        "gcc"                   # Param 6 – select compiler. Leaving it empty for automatic select.
```

Please keep in mind that this function shall process the print_status output and return the following exit codes:

(a) **0** – Successful. No error.
(b) **10** – not available. Recommended to skip or behave accordingly.
(c) **Non-zero** – error occurred.

AutomataCI runs its built sequences in parallel based on the hardware availability to speed things up. Hence, you only need to focus on what to compile and where to source your files.

#### 3.6.7.5.1 . File Structure Organization

AutomataCI designed the C file structure organization in a way similar to other modern Programming languages where the header files (*.h*), external object files (*.o*), external library files (*.a*) for a single module is can be packed under a single directory. Each header files represents a package in the module and shall be made importable by the *main.c* file.

Should any of the source file is a compiled object or a compiled library format, AutomataCI shall copy the compiled file over to the workspace during compilation phase for linking purposes.

For reference, this is the example:

```
srcC/
├── headers
├── libs
│   └── sample
│       ├── automataCI.txt
│       ├── entity_any-any.h
│       ├── greeter_any-any.c
│       ├── greeter_any-any.h
│       └── location_any-any.h
├── main-bin.txt
└── main.c
```

#### 3.6.7.5.2 . The Build List Text File

AutomataCI relies on a build list text file (replacing *Makefile*) that lists all the required files for the compilation and linking processes for your product. This built list shall be validated for producing a parallel control file and an linking list file to produce the final output. You're free to name this text file as long as it is being specified correctly in the *BUILD::compile* parameter.

Each line (separated by newline (*n*)) represents a source file either with:

1. ***.c*** extension – C source code requiring compilation to *.o* object file
2. ***.o*** extension – compiled object source file to be copied over

They are written in the following format:

**[TARGET_OS]-[TARGET_ARCH] [SETTING1] ... [SETTINGN] [RELATIVE_PATH]**

The simplest form is:

**[TARGET_OS]-[TARGET_ARCH] [RELATIVE_PATH]**

To support cross-platform in a single build, you're free to specify which source file should only be imported for the designated target os or target architecture. The value '***any***' signal AutomataCI that this source code shall always be imported regardless of the target system.

To facilitate better communications, comments are allowed and shall be led with the hash symbol (#).

Example file:

```
# DEPENDENCIES
any-any libs/sample/greeter_any-any.c

# CORE
any-any main.c
```

#### 3.6.7.5.3 . Build Sequences

AutomataCI uses the **$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP** directory to setup and build the workspace. The build sequence is as follows:

1 Parse the build list text file
  1.1 Validate source file existence
  1.2 Check the source file's compatibility (*.c* and *.o* file extensions only)
  1.3 Register the source file to:
    ***1.3.1 $PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/o-list.txt***
    ***1.3.2 $PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/sync.txt***
  1.4 Run its parallel build based on ***$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/sync.txt*** list.
  1.5 Link all object files based on ***$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/o-list.txt*** and export the final executable and linkable format (ELF) output or generate an *ar* type archived library file (*.a*).

The constructed command that permits library linking and integration is based on the following pattern:

***${compiler} -o ${output.o} -c ${source.c} ${arguments}***

Example:
*x86_64-linux-gnu-gcc -o {...}/lib/greeter_any-any.o -c {...}/libs/sample/greeter_any-any.c -Wall -Wextra -std=gnu99 -pedantic -Wstrict-prototypes -Wold-style-definition -Wundef -Wno-trigraphs -fno-strict-aliasing -fno-common -fshort-wchar -fstack-protector-all -Werror-implicit-function-declaration -Wno-format-security -pie -fPIE -Oz -static*

This command can be verified in the ***$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/sync.txt*** file for each build execution.

The arguments are intentional inserted last in case there is a need for specifying library (*.a*) files directory and linking a list of library files (see later section).

Any custom post-processing execution (e.g. AVR hex dump from the ELF file) task shall be executed in the custom .CI script after the compile function.

#### 3.6.7.5.4 . Libraries Management

For those who are interested to build a dynamically linked executable against an internal library, it's highly recommended to:

1. build the library as an external product first; AND THEN
2. The main executable to import it in as a source file; AND THEN
3. The main executable is then built with the library file.

Otherwise, it is strongly advised to build the main executable from all available source codes and statically link them up together.

**AutomataCI shall and always build the library file (*.a*) matching the AutomataCI build naming convention**. It's the Packaging CI job duty to export into the library file to a compliant name like *lib** prefix requirement in Linux OS system.

As AutomataCI facilitates compilers' option customization, you can construct your arguments in your build CI script with *-L* and *-l* option placing as last items among the list. Example:

```
...

SETTINGS_BIN="\
-Wall \
-Wextra \
...
-Oz \
-static \
-L/path/to/your/custom/libs \
-llibCustomA \
-llibCustomB
"


...

BUILD::compile "binary" "linux" "amd64" "main-bin.txt" "$SETTINGS_BIN" "$COMPILER"
if [ $? -ne 0 -a $? -ne 10 ]; then
	EXIT_CODE=1
fi
```

#### 3.6.7.5.5 . Cross-Compilation

While AutomataCI attempts to facilitate as much cross-compilation feature as possible in a simple approach for C programming language, there are limitations based on the availability in the host machine and the restrictions from the target system.

##### *3.6.7.5.5.1 . Linux (Debian-based)*

In summary, Linux Debian cross compilation facilities enhanced by brew installer choice is among the best among all known development host system.

##### *3.6.7.5.5.2 . Apple OSes (OSX and iOS)*

Although the POSIX side of stuffs (e.g. libraries, ABI, etc) are similar, Apple's SDK and C libraries are notoriously complicated to be made available in the non-Apple operating system. The Clang compilation is supported but *they are missing the Apple's SDK which then caused their cross-compilation to fail. Moreover, the binary notary security requirement further complicates the build process itself.*

Therefore, **like all Apple products, to build for Apple system, use a Mac OSX and its Xcode instead**.

Supports for Apple OSX to cross-compile for other OSes are limited compared to the Linux Debian counterparts (where it supports rare CPUs like *mips* and etc).

##### *3.6.7.5.5.3 . Windows*

C in Windows only allows MingW compiler to compile for its own kind. There are no records of any successful cross-compilation to other OSes.

To avoid complications, AutomataCI only facilitates MingW compilation for Windows using MingW only.

#### *3.6.7.6 . Go Programming Language*

When using the provided *srcGO/* directory, by default, AutomataCI deploys a pure static Go binary compilation by:

1. Disabling CGo (*CGO_ENABLED=0*).
2. Stripped all debugging symbols and trimpath.
3. PIE build mode if available (*-buildmode=pie*).

These output binary are fully static which can be operated even in an empty container (scratch).

However, this also comes at a cost: while AutomataCI loops through all the distribution list (*$ go tool dist list*), not all the target can be built without CGO. The list is available in the build_[SYSTEM] CI job script inside the *srcGO/* directory. This list shall be updated from time-to-time aligning to Go upstream updates.

#### *3.6.7.7 . Homebrew Ecosystem*

Homebrew (https://docs.brew.sh/) ecosystem is a rather complex that looks deceivingly simple ecosystem. While it offers a lot of ways to assemble the product for its customers including git cloning the entire repository, after careful analysis with all its developer's use cases permutations, the only sensible and sustainable way to proceed is packing its own archive package (in *tar.xz* format) containing a specific version's source code to build and let Homebrew to build it.

Hence, in order to support and enable Homebrew packaging, a placeholder file shall be created (either by touching or write an empty file) for Package Job to pick up the task later:

*myproc-homebrew_any-any*

#### *3.6.7.8 . Nim Programming Language*

For building the binary that are coded using Nim Programming Language, due to the nimbleness of the the language itself, AutomataCI by default, uses C build sequences and settings from the C Programming Language CI Build job as its default output.

Unlike other languages, Nim is first compile into a targeted source code (in our case: C source code) internally and then uses the native compiler to compile into the executable or library (in our case: using C CI build job functions). Hence, there are multiple forms of output per builds:

1. The executable
2. The compiled source code (*.c*, *.js*, *.objc*, ...)

##### 3.6.7.8.1 . Default Build Configurations

AutomataCI configures the default build configurations specifically matching the upstream Nim settings (e.g. ORC garbage collector with extreme release optimization mode) alongside C Programming Languages' build configurations. You should update it accordingly for your project. Please refer: https://nim-lang.org/docs/nimc.html for detailed info.

###### *3.6.7.8.1.1 . WASM Compilation*

At this point in time, WASM compilation cannot be confidently produce due to the absent of a proper Enscripten support (see: https://github.com/nim-lang/Nim/issues/8713). Until the support is ready, AutomataCI shall revisit and rebuild.

###### *3.6.7.8.1.2 . AVR Compilation*

At this point in time, AVR-GCC compilation failed in Nim while the source code and compilers are working in C Programming Language. It is suspected that the Nim standard libraries is at fault without factoring in Non-GC implementations and ecosystem (See: https://disconnected.systems/blog/nim-on-adruino/, https://github.com/zevv/nim-arduino, and https://github.com/dinau/nimOnAVR).

Hence, AutomataCI shall not support AVR compilation by default and considered Nim is incapable of facilitating Embedded development without extreme customization.

#### *3.6.7.8.1.3 . Nim's Docgen Auto-Documentation*

While Nim offers docgen capability to produce the package's documentations, at this point, it is not able to compose them properly without errors. Hence, documentations shall be done externally or source-codes driven manner.

### 3.6.7.8.2 . Distributing Source Code

It's vital to distinguish "source code" meaning in Nim Programming Language where in AutomataCI context, is referring to *the compiled source codes* AND *the Nim source codes*. Likewise, to distribute the source codes in a package, in this Build CI Job, you should create the corresponding *-source* flag file(s) for package CI job to pick up. In case you wish to distribute both types of source codes, you can create multiple flag files as such:

*myproc-source-c_any-any*
*myproc-source-nim_any-any*
*myproc-source-js_any-any*
*...*

**IMPORTANT**: as long as **the *-source* keyword is in the name**, Package CI job should be able to recognize it and acts accordingly.

### *3.6.7.9 . Python Programming Language*

For building the binary that are coded using Python Programming Language, 1 Python related dependency is required to compile the product into a single semi-statically linked binary:

1. pyinstaller

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.

#### 3.6.7.9.1 . Documentations

For composing the source codes' documentations, *pdoc3* pip package is used to generate the required HTML files. They can be published directly to the static site services like GitHub Pages.

#### 3.6.7.9.2 . Dependencies Installations

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!
pdoc3
coverage
pyinstaller
wheel
twine
```

### *3.6.7.10 . Rust Programming Language*

AutomataCI relies heavily on *cargo build* command to build the final executable based on the available optimization documented in the following resources:

1. https://doc.rust-lang.org/cargo/reference/profiles.html
2. https://doc.rust-lang.org/nightly/rustc/platform-support.html

#### 3.6.7.10.1 . Prioritized MUSL Static Binary Compilation

As per AutomataCI mantra to produce single binary product for end-users, AutomataCI always prioritizes *Musl* compilation for producing a static binary operating on its own. Unless otherwise required, customization can be done via the *build* or *materialize* CI job scripts.

#### 3.6.7.10.2 . Cross-Compilations

AutomataCI by default also supports cross-compilations based on the available compilers provided by the host system. The identified and supported cross-compiling targets are available in the following files:

*automataCI/services/compilers/rust.{sh,ps1}*

Specifically in these functions:

**RUST::get_build_target** (UNIX) || **RUST-Get-Build-Target** (Windows)

To procure a compiler, simply run the following command (can be added inside *Setup* CI job script) after setting Rust up entirely:

```
$ rustup target add x86_64-unknown-linux-musl
```

Due to the large amount of compiler choices in the catalog, AutomataCI only installs host's cross-compilers in the provided *srcRUST/* setup CI job directory.

### 3.6.8 . Notarize

Notarize job is responsible for code-signing all the applicable built executable for cryptographic authenticity purposes naming in Windows and Apple ecosystems. This CI job is based on the following documentations:

1. https://learn.microsoft.com/en-us/dotnet/framework/tools/signtool-exe
2. https://github.com/mtrojnar/osslsigncode
3. https://stackoverflow.com/questions/18287960/signing-windows-application-on-linux-based-distros
4. https://stackoverflow.com/questions/46649825/signtool-exe-equivalent-in-powershell

This is a special job where instead of being override by a custom CI job recipe, the custom CI job recipe supplies the required signing functions instead.

This job importing sequences is as follows, where the latter overwrites the former:

1. baseline job recipe (e.g. *src/.ci/notarize_unix-any.sh* via *PROJECT_PATH_SOURCE*); AND THEN
2. tech-specific job recipe (e.g. *srcPYTHON/.ci/notarize_unix-any.sh* via *PROJECT_PYTHON)*

**It's highly recommended to keep all the notarize CI job algorithms inside the baseline job recipe only. It's very rare that a tech-specific signing tool is required at all.**

Should the ***$PROJECT_SIMULATE_RELEASE_REPO*** is set, this job shall simulate the job completion without actually signing the file. This is facilitated mainly for external CI tools integration where leaking the sensitive credentials and files (e.g. cert and private keys) makes little sense.

The supported ecosystems are documented in the specific sub-sections.

#### *3.6.8.1 . Apple Ecosystems*

AutomataCI only supports Apple ecosystems notarization only in Apple's development machine (e.g. Mac Mini, Macbook, etc) as always required by Apple. Cross-notarization is not possible at all thanks to Apple's vendor locked-in restrictions. The documentations AutomataCI is based on are:

1. https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/customizing_the_notarization_workflow
2. https://stackoverflow.com/questions/56890749/macos-notarize-in-script
3. https://github.com/thezik/notarizer/blob/main/notarizer

##### 3.6.8.1.1 . Supported Platform

AutomataCI supports the following Apple ecosystem notarization for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Not Supported** | **Not Supported** | **Supported (Untested)** | **Not Supported** |

### *3.6.8.2 . Microsoft Ecosystems*

Microsoft ecosystems notarization is entirely based on the following documentations:

1. https://learn.microsoft.com/en-us/dotnet/framework/tools/signtool-exe
2. https://github.com/mtrojnar/osslsigncode

In Windows OS, the native *signtool.exe* signing executable is used instead of relying an external 3rd-party program.

In UNIX OSes however, a 3rd-party tool *osslsigncode* is used to only sign and notarize any Windows OS' cross-compiled executables.

To perform proper notarization, the following environment variables have to be supplied (either directly OR via *SECRETS.toml* file):

1. ***MICROSOFT_CERT*** – the full file-path to the cert file. On UNIX system, the cert format can be normal cert, SPC, or PCKS12 contained. On Windows however, only PCKS12 is accepted.
2. ***MICROSOFT_CERT_HASH*** – the required checksum algorithm specification. Can be: 'SHA256', 'MD5', 'SHA1', 'SHA2', 'SHA384', and 'SHA512'. When in doubt, use SHA256.
3. ***MICROSOFT_CERT_TYPE*** – the required cert type specification only in UNIX system. Values can be: 'CERT', 'SPC', and 'PCKS12'.
4. ***MICROSOFT_CERT_TIMESTAMP*** – the required Timestamp source usually an URL provided by a certificate authority.
5. ***MICROSOFT_KEYFILE*** – optional file used only in UNIX system when 'CERT' or 'SPC' cert types are used. It holds the private key of the certificate.
6. ***MICROSOFT_CERT_PASSWORD*** – The required password to decrypt the cert file.

#### 3.6.8.2.1 . Supported Platform

AutomataCI supports the following Microsoft ecosystem notarization for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

### 3.6.9 . 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 ecosystem, Red Hat's DEF 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.

This job importing sequences is as follows, where the latter overwrites the former:

3. baseline job recipe (e.g. *src/.ci/package_unix-any.sh* via *PROJECT_PATH_SOURCE*); AND THEN
4. tech-specific job recipe (e.g. *srcPYTHON/.ci/package_unix-any.sh* via *PROJECT_PYTHON)*

**To avoid overlapping implementations and chaotic duplication, it's highly recommended to keep all the package CI job algorithms inside the baseline job recipe only**. Although tech-specific CI job recipes are made available, they are meant for one to merge back into the baseline job recipes. Only the owner of the project knows what technologies to be used and AutomataCI shall not dictate but to facilitate all possible use cases only.

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}[*]_{OS}-{ARCH}[.{EXTENSION}]

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 next CI job: release.

#### *3.6.9.1 . Parallel Executions*

Due to the fact that it is possible for 1 product to produce a combinations of multiple ecosystems' packages, starting from version v1.7.0, AutomataCI supports parallel packaging executions whenever possible.

This makes packaging efforts faster for those independent packagers while still maintaining the series executions for those that are not possible. The total number of parallel executions available is determined by the total number of logical threads available in a host machine.

#### *3.6.9.2 . Cryptography Requirements*

It's duly noted that some ecosystems require cryptography notorization 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 directly like security vulnerabilities).

#### *3.6.9.3 . 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}/

#### *3.6.9.4 . Archive Packages (.tar.xz|.zip)*

AutomataCI supports primitive package archiving using **.tar.xz** or **.zip** archivers depending on the target OS (Windows is the only one using **.zip** archive format).

By default, AutomataCI deploys the maximum compression performance (e.g. level 9 for XZ compressor) to ensure the output can be stored in long-term storage elsewhere and be energy efficient during transit.

Starting from version 1.7.0, AutomataCI package *Archive* type in parallel execution.

##### 3.6.9.4.1 . Supported Platform

AutomataCI supports package archiving for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Supported** | **Supported** |

### *3.6.9.5 . Cargo Ecosystem (Rust)*

AutomataCI supports Cargo ecosystem when *PROJECT_RUST* is enabled and a placeholder file with *-cargo* keyword in its name is made available in the previous CI Build Job. AutomataCI supports Cargo ecosystem based on the following documentations:

1. https://doc.rust-lang.org/cargo/reference/registries.html
2. https://doc.rust-lang.org/cargo/reference/manifest.html
3. https://doc.rust-lang.org/cargo/reference/publishing.html
4. https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html

The goal is to assemble a healthy Rust crate for up-streaming to a cargo registry (e.g. https://crates.io). Thanks to the well-designed *Cargo* tool where it has an internal crate linting feature, AutomataCI shall use it to validate the assembled crate before package it for release.

The input placeholder file must have the following naming convention:

${PROJECT_SKU}**-cargo**_${__os}-${__arch}

Starting from version 1.7.0, AutomataCI package *Cargo* type in parallel execution.

#### 3.6.9.5.1 . Supported Platform

AutomataCI supports Cargo packaging for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Supported** | **Supported** |

#### 3.6.9.5.2 . Content Assembling Function

The content assembling function for UNIX OS is:

```
PACKAGE::assemble_cargo_content() {
	...
}
```

While the content assembly function for Windows OS is:

```
PACKAGE-Assemble-Cargo-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 objectives are:

1. to assemble a buildable source codes consumable by *Cargo* and its build tools; AND
2. to script the *${_directory}/Cargo.toml* crate file (specifically the **[package]** fields) based on the AutomataCI configurations used by *Cargo* externally at the end user side.

#### *3.6.9.5.2.1 . Required Files*

The following are the required files for AutomataCI to process the package properly:

1. *Cargo.toml* – template for generating the final *Cargo.toml* file.

The original *Cargo.toml* file (residing in *$PROJECT_RUST* directory) is used for operating Rust crate as usual with the **[package]** fields ignored. A special commented flag (**# [AUTOMATACI BEGIN]**) MUST be specified to tell AutomataCI to append anything lines after it during the *Cargo.toml* file generation.

The sole reason for regenerating *Cargo.toml* file during packaging is to make sure the **[package]** fields contain the *CONFIG.toml* project metadata values consistently across all other technological implementations.

AutomataCI shall *run Cargo* to validate the assembled package before staging it for Release CI job. Should there be any errors reported by *Cargo*, it should be resolved using your Rust programming language knowledge.

### *3.6.9.6 . Chocolatey Ecosystem*

AutomataCI assembles and scripts Chocolatey's package for distributing the product in the Chocolatey ecosystem based on the following documentations:

1. https://docs.chocolatey.org/en-us/create/create-packages
2. https://learn.microsoft.com/en-us/nuget/reference/nuspec
3. https://blog.chocolatey.org/2016/01/create-chocolatey-packages/
4. https://learn.microsoft.com/en-us/nuget/
5. https://learn.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-the-dotnet-cli
6. https://learn.microsoft.com/en-us/nuget/create-packages/creating-a-package
7. https://community.chocolatey.org/courses/creating-chocolatey-packages/summary-rules-and-guidlines
8. https://stackoverflow.com/questions/14329243/examine-contents-of-a-nuget-package
9. https://blog.chocolatey.org/2016/01/host-your-own-server/
10. https://docs.chocolatey.org/en-us/features/host-packages#local-folder-permissions
11. https://docs.chocolatey.org/en-us/features/host-packages#local-folder-or-share-structure

Chocolatey relies heavily to DotNet framework and some heavy batteries just to create a "*... a fancy version of a zip file that knows about metadata, versioning and dependencies related to underlying software, plus optional automation scripts that are run during installation, upgrade and uninstallation of the package (— by Chocolatey)*". Hence, AutomataCI itself uses the understanding of the Chocolatey & NuGet package algorithms and developed its own compiler for cross-packaging sake.

AutomataCI scripts all the required files and zip it accordingly. By default, *AutomataCI recommends only package a compilable source codes alongside AutomataCI tool for end-user's compilation*. This is mainly to handle 2 problems:

1. Microsoft Signing Security Requirement (https://learn.microsoft.com/en-us/windows/win32/seccrypto/signtool) — ensures the built binary is code-signed locally at the end user side of things; AND
2. Fulfilling Chocolatey's basic requirement.

Should the package delivers pre-compiled binaries, it's entirely your responsibility to get the binary both code-signed and notarized in your build system. Otherwise, should the product is accepted by the Chocolatey Core team, the package is published. *For GUI and etc, AutomataCI highly recommends to split it into a separate package since not everyone is using GUI interface*.

For self-signing on Windows, one can use SignTool in Microsoft Windows (https://learn.microsoft.com/en-us/windows/win32/seccrypto/signtool) or OSSLSignCode elsewhere (https://github.com/mtrojnar/osslsigncode). To ensure maximum compatibility and authenticity, Microsoft requires you to *purchase a trusted EV Certificate. Otherwise, self-signed CA certificate is acceptable.*

Starting from version 1.7.0, AutomataCI package *Chocolatey* type in parallel execution.

#### 3.6.9.6.1 . Supported Platform

AutomataCI supports Chocolatey package for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Supported** | **Supported** |

#### 3.6.9.6.2 . Content Assembling Function

The content assembling function for UNIX OS is:

```
PACKAGE::assemble_chocolatey_content() {
	...
}
```

While the content assembly function for Windows OS is:

```
PACKAGE-Assemble-CHOCOLATEY-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 containing a *Data/* sub-directory and a *tools/* sub-directory. The objectives are:

1. to assemble a build-able source codes consumable by Chocolatey and its build tools inside this *Data/* sub-directory; AND
2. to script the:
    1. *${_directory}/tools/chocolateyinstall.ps1;* AND
    2. *${_directory}/tools/chocolateyinstall.ps1*; AND
    3. *${_directory}/tools/chocolateyBeforeModify.ps1*

required Chocolatey PowerShell scripts.

#### *3.6.9.6.2.1 . Required Files*

The following are the required files for AutomataCI to process the package properly:

1. *${_directory}/tools/chocolateyinstall.ps1* – the install execution instructions in PowerShell format.
2. *${_directory}/tools/chocolateyinstall.ps1* – the uninstall execution instructions in PowerShell format.
3. *${_directory}/tools/chocolateyBeforeModify.ps1* – the pre-configurations (e.g. disabling services and etc) execution instructions before executing any install/uninstall PowerShell scripts above.

Should the required files are missing, AutomataCI shall fail the Package CI Job run.

#### *3.6.9.7 . 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.

Starting from version 1.7.0, AutomataCI package *.deb* type in parallel execution.

##### 3.6.9.7.1 . 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** | **Supported (Untested)** | **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.

Currently, due to the absence of Linux file permission (*chmod*) and *ar* program in native Windows system, packaging *.deb* is entirely at risk of packaging a malfunctioned one.

#### 3.6.9.7.2 . Content Assembling Function

The content assembling function for UNIX OS is:

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

While the content assembly function for Windows OS is:

```
PACKAGE-Assemble-DEB-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"
_changelog="$6"
```

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 sufficient utilities to create all the required files. It's entirely your duty to assemble all the files in their respective location and only create the **control/control** file as the last step due to its requirement of calculating **data/** directory disk space consumption.

#### *3.6.9.7.2.1 . 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.

#### *3.6.9.7.2.2 . Control File*

AutomataCI shall generate a control file (*control/control*) should its overriding version is unavailable using the *CONFIG.toml* data. In its generative product, AutomataCI also checks on the *Architecture* field's value to make sure it's compatible with its industrial-wide values (e.g. *x86_64* into *amd64*).

#### *3.6.9.7.2.3 . 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.

#### 3.6.9.7.3 . 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 }}
```

#### 3.6.9.7.4 . 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 your given 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.

#### 3.6.9.7.5 . 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}/docs/ABSTRACTS.txt

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

#### 3.6.9.7.6 . Closing the Debian's Apt Source.List Distribution Loop

By default, each packaged *.deb* package includes its own apt's *source.list* file via the *DEB::create_source_list* function. It also generates the necessary GPG key file required to authenticate any future *.deb* package update from the static repository upstream.

Hence, the customer only has to install any *.deb* package and just perform "*apt-get update*" or "*apt update*" to get the latest and greatest.

#### 3.6.9.7.7 . 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.

#### *3.6.9.8 . Docker*

Docker containerization is a promised cross-platform capable, horizontally scalable, and automated orchestrate-capable container packaging solution to answer massive reliable service needs. 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/
7. https://docs.docker.com/build/attestations/slsa-provenance/
8. https://github.com/orgs/community/discussions/45969
9. https://github.com/opencontainers/image-spec/blob/main/annotations.md#pre-defined-annotation-keys
10. https://docs.github.com/en/packages/learn-github-packages/connecting-a-repository-to-a-package
11. https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry
12. https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/
13. https://docs.github.com/en/packages/managing-github-packages-using-github-actions-workflows/publishing-and-installing-a-package-with-github-actions

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 full statically linked product packaging into a scratch container is given the highest priority for small-sized image**.

Due to the complexities made by Docker Builder especially dealing with multi-arch images construction, AutomataCI using its *buildx* component and disables its ATTESTATIONS capability for maximum OCI compatibility with other registries like GitHub Packages.

To ensure Docker build is smoothly executed, the local Docker builder **MUST logins into the targeted docker registry and push incompatible images (as in image platform is different from the host machine) directly**. To remove complexities, **AutomataCI uses *PLATFORM_VERSION*** tag format instead of the conventional *VERSION* or *latest* format.

In the end, AutomataCI shall:

1. Build all multi-arch Docker image stored remotely at registry directly.
2. Append the full distribution reference into the list file stored inside PROJECT_PATH_PKG directory for common manifest reference creation in release page.

Starting from version 1.7.0, due to Docker's limitation, AutomataCI package *Docker* type in series execution.

#### 3.6.9.8.1 . 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** | **Supported (Untested)** |

#### 3.6.9.8.2 . Open Container Initiative (OCI) Compatibility

To ensure the built images are available to as many containers' ecosystem as possible, AutomataCI heavily complies to OCI's engineering specifications and have Docker-specific metadata removed (build with *BUILDX_NO_DEFAULT_ATTESTATIONS=1* environment variable and *--provenance=false, --sbom=false* arguments) for the build commands.

The label '*org.opencontainers.image.ref.name*' is automatically filled in the build command via the argument (*--label "org.opencontainers.image.ref.name=${__tag}"*). Hence, manual filling in the Dockerfile is not required.

#### 3.6.9.8.3 . 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 **$_directory** variable should point to the workspace directory. The objective is to assemble all the required files and resources into this directory and generate the required *Dockerfile* (named as it is, see later section) specific to the target OS and CPU architecture. For example, the **$_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 disable this packaging task.
(b) **1** – Error is found.
(c) **0** – All good and proceed.

#### 3.6.9.8.4 . 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* (or *FS-Write-File* in PowerShell) function. A typical format is shown in the following page. Should the Dockerfile is missing, the Package CI job shall fail immediately.

To keep things minimal, the recommended (not a rule) base images are:

| OS | App Type | Recommended Images (FROM value) |
|---|---|---|
| Linux | Pure static | --platform=${_target_os}/${_target_arch} scratch |
| Linux | Dynamic | --platform=${_target_os}/${_target_arch} linuxcontainers/debian-slim:latest |
| Unknown | Pure static | --platform=${_target_os}/${_target_arch} scratch |
| Unknown | Dynamic | --platform=${_target_os}/${_target_arch} linuxcontainers/debian-slim:latest |
| Windows | Pure static | --platform=${_target_os}/${_target_arch} mcr.microsoft.com/windows/nanoserver:ltsc2022 |
| Windows | Dynamic | |
| Darwin | Pure static | **Not supported** |
| Darwin | Dynamic | **Not supported** |

To workaround of creating some required directory in the *scratch* type image, simply script a empty .tmpfile to the destination directory and copy into it. Example, to create */tmp* directory, the following instruction is used (assuming the .blank empty file is created):

```
COPY .blank /tmp/.tmpfile
```

Example of scripting a *Dockerfile* in the assembling function:

```
PACKAGE::assemble_docker_content() {
	...
	FS::write_file "${__directory}/Dockerfile" "\
# Defining baseline image
FROM --platform=${_target_os}/${_target_arch} scratch
LABEL org.opencontainers.image.title=\"${PROJECT_NAME}\"
LABEL org.opencontainers.image.description=\"${PROJECT_PITCH}\"
LABEL org.opencontainers.image.authors=\"${PROJECT_CONTACT_NAME} <$
{PROJECT_CONTACT_EMAIL}>\"
LABEL org.opencontainers.image.version=\"${PROJECT_VERSION}\"
LABEL org.opencontainers.image.revision=\"${PROJECT_CADENCE}\"
LABEL org.opencontainers.image.url=\"${PROJECT_CONTACT_WEBSITE}\"
LABEL org.opencontainers.image.source=\"${PROJECT_DOCKER_OCI_SOURCE}\"

# 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
	...
}
```

#### 3.6.9.8.5 . Required Configurations

As stated earlier, AutomataCI requires a number of environment variables in order to operate Docker properly. These are:

| Name | Provider | Purposes |
|---|---|---|
| PROJECT_DOCKER_REGISTRY | CONFIG.toml | Defines the registry's handle. |
| PROJECT_SOURCE_URL | CONFIG.toml | Defines the source code location. |
| CONTAINER_USERNAME | SECRETS.toml | Used for Docker Login function. |
| CONTAINER_PASSWORD | SECRETS.toml | Used for Docker Login function. |

### *3.6.9.9 . Homebrew Ecosystem*

AutomataCI assembles and scripts Homebrew's Formula Ruby script for distributing the product in the Homebrew ecosystem based on the following documentations:

1. https://docs.brew.sh/
2. https://docs.brew.sh/Taps
3. https://docs.brew.sh/Formula-Cookbook
4. https://github.com/Homebrew/homebrew-core/tree/master
5. https://brew.sh/2020/11/18/homebrew-tap-with-bottles-uploaded-to-github-releases/

Homebrew itself is somehow complicated when it comes to being automated by any CI pipelines mainly due to Apple's security requirements and its rather poor management over handling its vast coverage of development permutation possibilities. Moreover, the introduction of its own brewing terminologies makes the technical comprehension notoriously confusing albeit being flamboyant and fancy. After careful analysis over Homebrew's ecosystem, it's safer to let Homebrew to have its own dedicated package archive for distributions.

By default, *AutomataCI recommends only package a compilable source codes alongside AutomataCI tool for end-user's compilation*. This is mainly to handle 2 problems:

1. Apple Gatekeeping Security Requirement (https://support.apple.com/en-my/guide/security/sec5599b66df/web) - ensures the built binary is code-signed locally at the end user side of things; AND
2. Fulfilling Homebrew's Formula basic requirement.

For pre-compiled binaries (or in Homebrew's term: "*bottle*"), it's entirely your responsibility to get the binary both code-signed and notarized in your MacOS build system (Refer: https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/resolving_common_notarization_issues). Otherwise, should the product is accepted by the Homebrew Core team, the bottle is built and notarized using their system and certificates.

As for any other probable use cases such as distributing GUI and etc, *AutomataCI highly recommends to use macOS's Xcode and its ecosystems instead of Homebrew for sanity sake since Apply requires to you join their paid Developer Program regardless anyway.*

Starting from version 1.7.0, AutomataCI package *Homebrew* type in parallel execution.

#### 3.6.9.9.1 . Supported Platform

AutomataCI supports Homebrew packaging for a varieties of host OS system. The table below indicates the supporting nature across known OS in the market.

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Supported** | **Supported** |

#### 3.6.9.9.2 . Content Assembling Function

The content assembling function for UNIX OS is:

```
PACKAGE::assemble_homebrew_content() {
        ...
}
```

While the content assembly function for Windows OS is:

```
PACKAGE-Assemble-HOMEBREW-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 objectives are:

1. to assemble a buildable source codes consumable by Homebrew; AND
2. to script the *${_directory}/formula.rb* Homebrew formula script used as a foundation for creating the finalized Homebrew formula Ruby source code.

#### *3.6.9.9.2.1 . Required Files*

The following are the required files for AutomataCI to process the package properly:

1. *formula.rb* – template for generating the Homebrew Formula Ruby source code.

The *formula.rb* can have the following placeholders (exactly as it is) for value replacement during the packing process:

1. **{{ TARGET_PACKAGE }}** - the tar.xz package name for AutomataCI to replace in; AND
2. **{{ TARGET_SHASUM }}** - the required SHA256 value of the tar.xz package.

Otherwise, its format complies strictly to Homebrew's Formula Cookbook (https://docs.brew.sh/Formula-Cookbook).

Should the required files are missing, AutomataCI shall fail the Package CI Job run.

### *3.6.9.10 . Open-Source Package (.ipk OR .opk)*

AutomataCI developed its own package compiler based on the Open-Source package (or commonly known as *.ipk*) information available here:

1. https://raymii.org/s/tutorials/Building_IPK_packages_by_hand.html
2. https://downloads.openwrt.org/
3. https://nilrt-docs.ni.com/opkg/opkg_intro.html
4. https://yairgadelov.me/custom-opkg-repository/
5. https://git.yoctoproject.org/opkg/

IPK is the actual predecessor from the Debian DEB package where it is heavily used in embedded worlds like OpenWRT and Yocto. Unlike Debian DEB that emphasize of OS unification, IPK is extremely aggressive towards package size and limited technological usage. Among known differences are:

1. Outer most layer uses *tar* tool as the archiver instead of *ar* tool; AND
2. Only needs 1 required *control/control* file instead of many checksum files; AND
3. No maintained and documented specifications.

Due to these differences and Windows is now supporting *tar* archiver and *gz* compressor (See: https://techcommunity.microsoft.com/t5/containers/tar-and-curl-come-to-windows/ba-p/382409), IPK becomes the most versatile and cross-platform friendly package across OSes.

Although AutomataCI uses its own compiler, the output shall always be compatible with common patterns and Yocto ecosystem via using *gz* compressor. However, you're still required to learn through the Debian specifications (at least binary package) shown above as a side reference before proceeding to construct the job recipe. Likewise, AutomataCI prioritizes binary package build since the source codes are usually distributed using *git* ecosystem.

Starting from version 1.7.0, AutomataCI package *.ipk* type in parallel execution.

#### 3.6.9.10.1 . 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** | **Supported (Untested)** | **Supported** | **Supported** |

#### 3.6.9.10.2 . Content Assembling Function

The content assembling function for UNIX OS is:

```
PACKAGE::assemble_ipk_content() {
        ...
}
```

While the content assembly function for Windows OS is:

```
PACKAGE-Assemble-IPK-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 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 sufficient utilities to create all the required files. It’s entirely your duty to assemble all the files in their respective location and only create the **control/control** file as the last step due to its requirement of calculating **data/** directory disk space consumption.

#### *3.6.9.10.2.1 . Required Files*

As specified by known practices and blogs, there are the known required files:

1. *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.

#### *3.6.9.10.2.2 . Control File*

AutomataCI shall generate a control file (*control/control*) should its overriding version is unavailable using the *CONFIG.toml* data. In its generative product, AutomataCI also process the given on the *Architecture* field's value instead of strict checking for flexibility sake.

#### *3.6.9.10.2.3 . 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.

#### 3.6.9.10.3 . Testing Packaged IPK's Health

To test the packaged IPK's health, simply use *tar* tool to unpack the package and verify accordingly.

Keep in mind that due to the outer archive layer is using *tar* instead of *ar* tool, Debian's *deb-dpkg* may not work properly.

### *3.6.9.11 . 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

Starting from version 1.7.0, AutomataCI package Flatpak type in series execution.

#### 3.6.9.11.1 . Supported Platform

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

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Not Supported** | **Not Supported** |

#### 3.6.9.11.2 . 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 10***" is suffice to inform the shell that its does nothing.

#### *3.6.9.11.2.1 . 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 AutomataCI 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

### 3.6.9.11.3 . 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**".

#### 3.6.9.11.4 . 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.

#### 3.6.9.11.5 . 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.

#### 3.6.9.11.6 . 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.

#### 3.6.9.11.7 . 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
```

#### 3.6.9.11.8 . Release to Repository

Due to *Flatpak-Builder*'s all-in-one capabilities, the Release Job is simultaneously executed in this stage. AutomataCI exports a private repo directory at:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/flatpak

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

#### 3.6.9.11.9 . Single Bundle Export

To ensure a fail-safe is available, AutomataCI automatically exports the single bundle format (https://docs.flatpak.org/en/latest/single-file-bundles.html) that allows user to manually import the software without needing to track a private repo. This bundle file is included in the PROJECT_PATH_PKG directory ending with *.flatpak* file extension as required.

#### *3.6.9.12 . 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.

Starting from version 1.7.0, AutomataCI package *.rpm* type in parallel execution.

##### 3.6.9.12.1 . Supported Platform

AutomataCI can only build rpm output in the following build environment

| UNIX (Debian based) | UNIX (Red Hat based) | MacOS | Windows |
|---|---|---|---|
| **Supported** | **Supported (Untested)** | **Not 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.

#### 3.6.9.12.2 . 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**.

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/${PROJECT_SKU}/
install -m 0644 copyright %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/

install --directory %%{buildroot}/usr/local/share/man/man1/
install -m 0644 ${PROJECT_SKU}.1.gz %%{buildroot}/usr/local/share/man/man1/
" >> "${__directory}/SPEC_INSTALL"
```

#### 3.6.9.12.3 . Required Files

AutomataCI requires the following files to perform a successful build:

1. *${_directory}/SPEC_INSTALL*
2. *${_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

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"
```

#### 3.6.9.12.4 . 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*

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

If the **${_directory}/SPEC_DESCRIPTION** spec fragment file is not provided, AutomataCI shall automatically parse and process data from the **${PROJECT_PATH_RESOURCES}/docs/ABSTRACTS.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.

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

#### 3.6.9.12.5 . Overriding The Entire Spec File

To override the spec file completely, simply create a fully compliant 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.

#### 3.6.9.12.6 . 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/

#### 3.6.9.12.7 . 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}/docs/ABSTRACTS.txt

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

#### 3.6.9.12.8 . 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.

#### 3.6.9.12.9 . Built-In Definitions

To ensure the packaging does not perform any alterations to the end product, AutomataCI deploys the following definitions by default to prevent *rpmbuild* from performing magical changes:

```
$ rpmbuild \
        --define "_topdir ${__directory}" \          # only work in workspace; not elsewhere
        --define "debug_package %{nil}" \            # ensure not to alter any packaged product.
        --define "__strip /bin/true" \               # ensure not to alter any packaged product.
        --target "$__arch" \
        -ba "$__.spec"
```

#### 3.6.9.12.10 .Closing the Red Hat's YUM Source.Repo Distribution Loop

By default, each packaged *.rpm* package includes its own yum's *source.repo* file via the *RPM::create_source_repo* function. It also generates the necessary GPG key file required to authenticate any future *.deb* package update from the static repository upstream.

Hence, the customer only has to install any *.rpm* package and just perform "yum *update*" to get the latest and greatest.

#### 3.6.9.12.11 .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.

### *3.6.9.13 . PyPi Library Module (Python)*

AutomataCI supports PyPi library module construction through the use of Python '*build*' and `*twine*` modules alongside `*setuptools*` module internally. In order to make sure there is a full compliance with Python, '*build*' and '*twine*' modules 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://packaging.python.org/en/latest/specifications/
2. https://packaging.python.org/en/latest/specifications/declaring-project-metadata/
3. https://peps.python.org/pep-0660/
4. 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.

For PyPi library packaging, it is noted that the documentation relies on **$PROJECT_PYPI_README** and **$PROJECT_PYPI_README_MIME** to define the package's public facing documentation set in *CONFIG.toml* configuration file. **This file MUST be copied into the assembling directory** during the content assembling function due to the sandbox access restriction effect caused by packaging virtual environment.

Starting from version 1.7.0, AutomataCI package *PyPi* type in parallel execution.

#### 3.6.9.13.1 . Supported Platform

Should the modules are 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** |

#### 3.6.9.13.2 . 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 *build* module and Python to construct the library package. Then finally, the function scripts the required **pyproject.toml** 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 disable this package task.
(b) **1** – Error is found.
(c) **0** – All good and proceed.

#### 3.6.9.13.3 . Required Files

As specified by the Python Packaging specifications, the following files are required:

1. **pyproject.toml**
2. your library source codes assembled in your end-user's importing manner
3. **README file defined in $PROJECT_PYPI_README** named exactly as defined by the variable.

Should the *README* file is missing or being renamed as something else (hence, not found), AutomataCI shall throw an error and stop the packaging job entirely.

Should *pyproject.toml* 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**.

#### 3.6.9.13.4 . Testing Package

`*twine*` pip module can be used for testing the generated packages using the following command:

```
twine check "${__directory}/dist/"*
```

If things are fine, you should get a log as follows:

```
Checking
pypi_automataci-src_1.5.0_darwin-amd64/dist/AutomataCI-1.5.0-py3-none-any.whl: PASSED
Checking
pypi_automataci-src_1.5.0_darwin-amd64/dist/AutomataCI-1.5.0.tar.gz: PASSED
```

### 3.6.10 . 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.

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.

This job importing sequences is as follows, where the latter overwrites the former:

1. baseline job recipe (e.g. *src/.ci/release_unix-any.sh* via *PROJECT_PATH_SOURCE*); AND THEN
2. tech-specific job recipe (e.g. *srcPYTHON/.ci/release_unix-any.sh* via *PROJECT_PYTHON)*

**It's highly recommended to keep all the release CI job algorithms inside the baseline job recipe only**.

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}[*]_{OS}-{ARCH}[.{EXTENSION}]

AutomataCI detects all the known packages located inside the directory defined in *CONFIG.toml*: the **${PROJECT_PATH_ROOT}/${PROJECT_PATH_PKG}** directory 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. At this point, should the customizable package-processor function is made available, the currently processing package is fed to that function. Once the entire looping is done, it shall runs all the associated technologies' post-processor functions. Upon completion, the content inside the same directory is ready for any remaining manual upload (e.g. GitHub Release).

#### *3.6.10.1 . 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**.

#### *3.6.10.2 . Special Custom Implementations*

Unlike all other jobs, Package Job recipe **requires a compulsory CI job recipe** to supply the required package content assembly functions. Example:

**RELEASE::run_post_processor() { ... }**
**RELEASE::run_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.

#### *3.6.10.3 . 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.

#### *3.6.10.4 . 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}/

#### *3.6.10.5 . Archive (.tar.xz | .zip)*

AutomataCI does not do anything for Archive packages in Release Job since its Package Job already completed all its tasks.

#### *3.6.10.6 . Cargo*

AutomataCI automatically login into the Cargo registry and publish the packaged cargo directory. The directory must having the following naming convention and **MUST** be a directory containing a releasable version of the Rust crate's files:

**cargo**_${PROJECT_SKU}-cargo_${__os}-${__arch}

AutomataCI relies on the following configurations perform the operations:

| **Name** | **Supplied by** | **Purpose** |
|---|---|---|
| CARGO_REGISTRY | CONFIG.toml | Registry location. Default is '**crates-io**'. |
| CARGO_PASSWORD | SECRETS.toml | The secret password or the secret API token provided by the registry. |

Upon successful publication, the package directory shall be deleted by AutomataCI automatically.

#### *3.6.10.7 . Changelog*

AutomataCI automatically seals the 'latest' changelog *data* and *deb* entries to the running **$PROJECT_VERSION** value. You should increase your project version number once the job is completed.

### *3.6.10.8 . Chocolatey*

AutomataCI uses *Git* technology to clone the given **$PROJECT_CHOCOLATEY_REPO** Git URL and export the nupkg package into its **Packages/** directory. Since Chocolatey does not have any kind of authenticity management system, a local file-based directory with read-only access (e.g. static hosting) similar to archive output is suffice.

A valid nupkg package MUST have:

1. the word *-chocolatey* in its name (usually a suffix to the project sku name); AND
2. ends with *.nupkg* file extension.

A good example would be:

automataci-chocolatey_1.6.0_any-any.nupkg

This package file must be made available in:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_PKG}/

#### *3.6.10.9 . Citation (CITIATON.cff)*

AutomataCI natively generates the *CITATION.cff* file in the Release CI job permitting the academic industry to properly reference the project or product. It's based on the following documentations:

1. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-citation-files#citation-formats
2. https://github.com/citation-file-format/ruby-cff
3. https://github.com/citation-file-format/citation-file-format/tree/main
4. https://citation-file-format.github.io/

By default, AutomataCI generates one inside the **$PROJECT_PATH_ROOT/$PROJECT_PATH_PKG/** directory against the release version. Should it be successful, a copy shall be exported to the **$PROJECT_PATH_ROOT/** directory where some service providers (e.g. GitHub) will parse the file for metadata processing of the project repository.

##### 3.6.10.9.1 . Abstract Customization

By default, AutomataCI seeks the common file located at:

**$PROJECT_PATH_ROOT/$PROJECT_PATH_SOURCE/docs/ABSTRACTS.txt**

for construct the *abstract:* data field. This same file is also used elsewhere for maintaining consistency across many ecosystems.

##### 3.6.10.9.2 . Citation Appendix

By default, AutomataCI seeks a YAML data file located at:

**$PROJECT_PATH_ROOT/$PROJECT_PATH_SOURCE/docs/CITATIONS.yml**

for appending additional fields there are specific to *CITATION.cff* file. There are fields that are generated automatically by AutomataCI. Anything else found in the specified file shall be appended as it is. Please keep in mind that this is a YAML file and contains strict manifest fields.

### *3.6.10.10 . 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}/static/deb

where it is the **{PROJECT_STATIC_REPO}** directory. The original package file is left in-tact in case some customer wants to install manually.

#### 3.6.10.10.1 .Reprepro Distribution Configuration File

AutomataCI automatically generates the required reprepro's *conf/distributions* file based on the *CONFIG.toml* data and said file is only being used to assemble all the latest *.deb* packages in an apt repository. The data file can be analyzed from the following directory:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/deb/

#### 3.6.10.10.2 .Supported Architectures

AutomataCI uses a combinations list of OSes and CPU architectures listed from "*dpkg-architectures -L*" command (e.g. *amd64* CPU architecture with *linux* and *openbsd* OSes yield "*amd64 linux-amd64 openbsd-amd64*" as a result.

These OS and CPU Architecture values are updated from time to time based on Debian stable OS until a suitable approach is found. The current lists are available in *automataCI/services/publishers/reprepro.[EXTENSION]* library.

#### 3.6.10.10.3 .Reprepro Database

AutomataCI always purge all repository's directory before using *reprepro* create a new one. Hence, retaining its database data are not sensible so therefore, AutomataCI houses them in the following directory:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/deb/db

### *3.6.10.11 . Docker*

AutomataCI assemble the '*latest*' and '*PROJECT_VERSION*' tag via the Docker manifest management in Release Job recipe. The supported documents are as follows:

1. https://docs.docker.com/engine/reference/commandline/manifest/

Upon the completion of the task:

1. The '*latest*' tag has been updated.
2. A generic multi-arch image '*VERSION*' tag is created or updated.

### *3.6.10.12 . Docs Repo*

AutomataCI detects the Docs repo directory (**${PROJECT_PATH_ROOT}/${PROJECT_PATH_DOCS}**) existence and shall only update and publish the content if exists.

Similar to static repo, AutomataCI uses docs repo instead. See *CONFIG.toml* for all available configurations. The main difference is that the content in the docs repo shall always be a static website filesystem readily available for publications.

### *3.6.10.13 . Homebrew*

AutomataCI uses *Git* technology to clone the given **$PROJECT_HOMEBREW_REPO** Git URL and export the formula ruby file into its **Formula/** directory. The **$PROJECT_HOMEBREW_REPO** is setup as **${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/$PROJECT_HOMEBREW_DIRECTORY}** directory for one to browse and manage.

A valid formula ruby file must have:

3. the word *-homebrew* in its name (usually a suffix to the project sku name); AND
4. ends with *.rb* file extension.

A good example would be:

automataci-homebrew_1.6.0_any-any.rb

This formula file must be made available in:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_PKG}/

Upon successful validation, AutomataCI shall copy it over to the Homebrew git repository and shall commit+push the changes at the end of the conclusion. The formula file remains in-tact for redundancy sourcing since Homebrew allows installation via formula script.

### *3.6.10.14 . Open-Source Package (.ipk OR .opk)*

Due to the lack of documentations + both OpenWRT and Yocto Project are using their respective derived *opkg* packages repository generator algorithm, there are no-way to properly and securely authenticate the package hosting.

At this point in time, AutomataCI release the *.ipk* packages like an archive package until an unified *opkg* specification is created.

### *3.6.10.15 . Red Hat Flatpak (Flatpak)*

AutomataCI does not do anything for Flatpak in Release Job since its Package Job already completed all its tasks.

### *3.6.10.16 . Red Hat Package (.rpm)*

AutomataCI uses *createrepo_c* external technology (https://github.com/rpm-software-management/createrepo_c) to process the *.rpm* Red Hat package into an yum friendly repository for customers to deploy using the famous “yum install” command.

Due to the requirement of *createrepo_c*, this release job function **can only work in Linux environment only**.

The destination is set to:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/rpm

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.

### *3.6.10.17 . PyPi Library Module (Python)*

AutomataCI uses pip installed *twine* program to upstream any compatible and detected PyPi library packages located in **${PROJECT_PATH_ROOT}/${PROJECT_PATH_PKG}** directory. The package **MUST** comply to the following conditions:

1. Is a directory; AND
2. Is housing a *.whl* (zip format) archive and a *.tar.gz* archive; AND
3. The directory name must lead with '**pypi**'.

#### 3.6.10.17.1 .PyPi Registry Account

The latest PyPi registry requires one to sign-up an account for upstream packages. It is duly noted that the private token is required and generated from your account. The secret token acts as the TWINE_PASSWORD while the TWINE_USERNAME is locked to '*__token__*' as its value.

In short, you have to supply 2 secret environment variables either by declaring them directly or via *SECRETS.toml* file:

1. **TWINE_USERNAME** – username instructed by the package registry (usually '*__token__*').
2. **TWINE_PASSWORD** – private token issued by the package registry.

#### 3.6.10.17.2 .PyPi Registry URL

One can also define a custom registry URL for PyPi in the *CONFIG.toml* via the environment variable: **${PROJECT_PYPI_REPO_URL}**. Recommended values are:

1. Test Zone : https://test.pypi.org/legacy/
2. Actual : https://upload.pypi.org/legacy/

#### 3.6.10.17.3 .Existing Package Failure Notice

PyPi does not play well with overriding existing package. Hence, you need to remove the existing package from the package before performing the release. Otherwise, Release job will fail. If you need to workaround, simply remove the pypi package directory manually.

### 3.6.11 . Stop

Stop CI job recipe is responsible for removing all the configurations setup by the Start CI job recipe, effectively stopping the development environment in the terminal. Usually, they are on-screen notices so please follow the instructions as presented.

Example, for Python programming language, it provides the instruction where to deactivate the Python Virtual Environment (*venv*).

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/stop_unix-any.sh* via *PROJECT_PYTHON)*; AND THEN
2. baseline job recipe (e.g. *src/.ci/stop_unix-any.sh* via *PROJECT_PATH_SOURCE*)

### 3.6.12 . Deploy

Deploy Job is a common task to facilitate the deployment of the new release into the ecosystems like Kubernetes controls and etc. By default, AutomataCI leaves the executable scripts empty to facilitate any changes.

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/deploy_unix-any.sh* via *PROJECT_PYTHON*); AND THEN
2. baseline job recipe (e.g. *src/.ci/deploy_unix-any.sh* via *PROJECT_PATH_SOURCE*)

### 3.6.13 . Clean

Clean Job is to remove all specified files and artifacts.

This job executes its job executions recipes as follows, in sequences:

1. tech-specific job recipe (e.g. *srcPYTHON/.ci/clean_unix-any.sh* via *PROJECT_PYTHON*); AND THEN
2. baseline job recipe (e.g. *src/.ci/clean_unix-any.sh* via *PROJECT_PATH_SOURCE*)

### 3.6.14 . Purge

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

| |
|---|
| **BE WARNED**: This is **a nuclear button**. Once nuked, you will need to setup everything including tooling from scratch. |

# 4 . Contribute to AutomataCI

Thank you for your interest in contributing towards AutomataCI source codes. Please keep in mind that this section is entirely different from building your own AutomataCI project-specific job recipes. All the code contributions here shall be upstream and be unified in the *automataCI/* directory which is distributed downstream for future upgrades.

## 4.1 . Tech Requirements

AutomataCI is fundamentally re-constructed from Go Programming Language into using only *POSIX Shell* and *Powershell* which can be operated natively without requiring additional installations. To contribute, you need a very deep understanding about them as AutomataCI always cross-translate them to make sure its functionalities are working on as many systems as possible.

### 4.1.1 . POSIX Shell

POSIX compliant shell (not to be confused with BASH shell) is an UNIX execution scripting language **meant for UNIX operating systems like Linux – Debian, BSD – Debian, OSX – Mac, and etc**.

To be POSIX compliant, you must first understand and familiarize yourself with BASH scripting and then works yourself towards POSIX compliant shell. Things like array, lowercase string changes, double square quote comparison (*[[ ... ]]*) are not available in POSIX shell.

Currently, AutomataCI prioritize linux and OSX using POSIX compliant shell. If you're new, reading the codes from the *automataCI/services* libraries can provide some good production-grade shell codes for you to learn and deploy in your future pursue.

### 4.1.2 . PowerShell

**Powershell is solely used for Windows OS** despite Microsoft permits its installation elsewhere. AutomataCI prohibits the latter for the fact since the installation requires Microsoft telemetry feature to be installed which breaks a perfectly fine UNIX operating system. Unlike OSX in Mac ecosystem, to date, *Microsoft Windows is still incompatible with POSIX shell without Linux subsystem (which is Linux VM-like anyway)*.

## 4.2 . Coding Styles

This section covers the coding styles that is compatible for both shells types and its rationale behinds them.

### 4.2.1 . Understanding The Shelling Nightmare

The most headache problem is to make both Powershell and POSIX Shell same pattern and same context like a human language i18n translation feature. It's like trying to marry water with oil working in the same way: *they don't mix*.

Powershell, like oil, is notoriously confusing, sticky, and complicated by itself. In POSIX shell, all commands only has to check against $? variable that contain last execution status. Powershell however, has too many. Here's a case:

```
# POSIX
some_command "..."
if [ $? -ne 0 ]; then
   exit 1
fi
```

versus

```
# POWERSHELL TYPE 1 (4 possibilities in a single command)
$__output = Invoke-Expression "some_command `"...`""
if (($LASTEXITCODE -ne 0) -or ($? -ne 0) -or (-not $__output)) {
   exit 1
}

# POWERSHELL TYPE 2 (another possibility via object method)
$__process = Start-Process `
        -Wait `
        -FilePath "$__program" `
        -NoNewWindow `
        -ArgumentList "$__arguments" `
        -PassThru
if ($__process.ExitCode -ne 0) {
        exit 1
}
```

PowerShell has its own innovative definitions that are completely incompatible with POSIX shell like interpreting double colon (::) for naming function. Example:

```
# POSIX
OUTPUT::function_name() {
	...
}
```

versus

```
# POWERSHELL
function OUTPUT-Function-Name {
	...
}
```

The worst of all – Powershell's *return* is not behaving as commonly expected. In POSIX shell, *return* always set the numeric value for *$?* variable. Powershell returns everything inside the function output so you need to redirect every single command's output or status values into $null in order to get a clean *$?* similar pattern.

```
function OUTPUT-Function-Name {
	Invoke-Expression "dir"
	return 0
}

$__output = OUTPUT-Function-Name (Get-Location)
# $__output is not 0; it's the $(dir output) + 0. Also, 0 is not a return code but is an output.
```

Remember the fact that AutomataCI prohibits Powershell installation in UNIX OS earlier? This is also because not all cmdlets are available if you do so. Journeying into this path makes creating a cross-platform Powershell script notoriously complicated by its guessing and gambling with fate nature. It's cute to find out Powershell aspiring to outgrow itself to become a (comparable?) full-fetch programming language but lack of architecture planning quite a bizarre.

Regardless, through many failed attempts, AutomataCI discovered a way to do it successfully as long as the scripts developers adhere to certain rules set by AutomataCI dealing with these arcane but powerful technologies.

### 4.2.2 . Testing Strategies and Countermeasures

To build a compatible pipelines for both water (POSIX Shell) and oil (PowerShell) while not impeding their respective innovation path line. Understand these business objectives clearly:

1. The paying customers don't pay for how great AutomataCI is; it's just an infrastructure to build the product that the paying customers are actually using; AND
2. Both POSIX shell and PowerShell are not a compilation kind of scripting language. They relies on line-by-line executions; AND
3. Ensure the control path of the executions are small enough that are highly reusable so that errors can be probed and observed by multiple uses, thus making it harder to cause an error; AND
4. AutomataCI is expected to deliver results in a 100% accuracy manner: either 100% correct or 100% error. There are no in-between probability like an AI product.

To facilitate these business objectives, the following sub-sections are to be observed and obeyed carefully when contributing to AutomataCI libraries and source codes.

#### *4.2.2.1 . Functionalize Everything*

To make sure the executions are re-usable (which helps in testing the codes themselves), all executions are to be completely functionalized complying to the following pattern:

```
# POSIX
LIBRARY::function_name_in_lowercase() {
	__param_name_1=”$1”
	...

	# validate input
	...

	# execute
	...

	# return status
	return 0
}
```

```
# POWERSHELL
LIBRARY-Function-Name-In-Dashing-Titlecase {
	param(
		[string]$__param_name_1,
		...
	)

	# validate input
	$null = ...

	# execute
	$null = ...

	# return status
	return 0
}
```

That way, when used, the calling for both types of shell are having close similarities:

```
# POSIX
LIBRARY::function_name_in_lowercase "..." "..." ...
if [ $? -ne 0 ]; then
        return 1
fi

...
```

```
# POWERSHELL
$__process = LIBRARY-Function-Name-In-Dashing-Titlecase "..." "..." ...
if ($__process -ne 0) {
        return 1
}

...
```

The rules:

1. POSIX shell should comply to its *lowercase_snake_case* function name while POWERSHELL should comply to its *Dashing-Titlecase* function name.
2. For POWERSHELL, remember to redirect all unused command execution to *$null* to make sure *return* is actually returning the exit code only.
3. For suite-function differentiation, POSIX Shell uses double colon (*::*) while PowerShell uses dash (-).
4. For suite naming, it shall be *UPPERCASE*.
5. Stick to positional parameters ***ONLY***. POSIX shell does not have any alternative so **stick to it.**
6. **[NON-COMPROMISING RULE]** Both POSIX and POWERSHELL must share maximum similarity to the point where it can be nearly 1:1 comparing to each other.
7. **[NON-COMPROMISING RULE]** All variables MUST be inserted as a quoted string.
8. **[NON-COMPROMISING RULE]** Maintain absolute silence at all time and only respond via return code.

In any cases, try to refer other services' scripts when stuck first. Ask if you may.

#### 4.2.2.1.1 . Output Data Processing Function

For output data processing function (e.g. lowercasing a string), **DO NOT** return the exit code since POWERSHELL cannot programmatically do so. Instead, choose a default value to return as negative expectation and have it checked instead. Example:

```
# POSIX
STRINGS::to_lowercase() {
	#__content="$1"

	printf "$1" | tr '[:upper:]' '[:lower:]'
}
```

```
# POWERSHELL
function STRINGS-To-Lowercase {
	param(
		[string]$__content
	)

	return $__content.ToLower()
}
```

Call them would be:

```
# POSIX
__output="$(STRINGS::to_lowercase "$__data")"
if [ "$__output" = "" ]; then
	return 1
fi
```

```
# POWERSHELL
$__output = STRINGS-To-Lowercase "${__data}"
if ($__output -eq "") {
	return 1
}
```

#### 4.2.2.1.2 . Simplifying POSIX Shell's Function Parameters

If the function is too simple and short (like the *STRING::to_lowercase* above), you're allowed to comment out the parameter naming and use the positional parameter in its original numbering instead.

The condition to apply this rule can be any of the following:

1. Only use 1 parameter; OR
2. Only use maximum of 2 parameters AND within a maximum of 10 lines long with max 3-tab complexities.

Otherwise, stick to declaring function naming. Examples:

```
# POSIX
STRINGS::to_lowercase() {
	#__content="$1"

	printf "$1" | tr '[:upper:]' '[:lower:]'
	return 0
}
```

```
# POSIX
STRINGS::has_suffix() {
	#__suffix="$1"
	#__content="$2"

	# execute
	case "$2" in
	*"$1")
		return 0
		;;
	*)
		return 1
		;;
	esac
}
```

#### *4.2.2.2 . Naming Conventions*

Fortunately, both shell types share the same limitations for both their functions and variables naming conventions. Understand that both shells do not have scoping capabilities so any constant, public, private scoping interpretation is strictly through naming convention.

All functions shall be re-importable as the same definitions over and over again so the naming convention plays a vital roles in libraries organization (see later sections).

In short, you **SHALL** do the following for both functions and variables in both shell types:

1 Comply to the lead underscoring naming convention to indicates its deepness:
   1.1 **Level 0 lowercase** – indicates an exportable entity (e.g. *create_this*, *Create-This*, *$pkg*)
   1.2 **Level 0 UPPERCASE** – indicates an exportable "constants" or a library suite (e.g *STRINGS::fx_name*, *STRINGS-Fx-Name*, *$PROJECT_PATH_ROOT*)
   1.3 **Level 1 lowercase** – indicates private entity (e.g. *_run_this_subroutine*, *_Run-This-Subroutine*, *$_target*)
   1.4 **Level 2 lowercase** – indicate private and frequently disposable entity (e.g. *__run_this_internal_function*, *__Run-This-Internal-Function*, *$__ret*)

2 Deepness level **SHALL NOT go beyond level 3**. If you do, it signifies you have a problem with re-organization (see next section).

AutomataCI is an infrastructure code. *Please DO NOT expect a lot of privileged resources like "generating a separate spectacular and fancy HTML documents and etc" natively*. Hence, **write in a way that folks will read the source code only (cake). Anything else are unwanted costs (diabetic icing). Focus on the cake, not complicating it with too much icing.**

**Write in full English** for native system compatibility where **Hungrian Notation is prohibited (e.g: *$__target* instead of *$__obj*)**. Write in a way that the maintainers only need to scroll vertically, top-to-bottom; not both directions. **Keep the width 80 characters max if possible while 100 is the hard limit for maximum terminal compatibility (including SSH terminals)**.

When reading your function, all executions are always visually traceable just by reading alone without any debugger or IDE assistance.

#### *4.2.2.3 . Organization*

With the everything functionalized approach and naming convention in place, it's time to speak the rules of organization. AutomataCI is being executed as follows, in layers, from top to bottom:

| Level | File | Trigger | Descriptions |
|---|---|---|---|
| **0** | ci.cmd | *Execute* | The Polygot executable script to unify user commands independent of OS and CPU architecture. |
| | automataCI/ci.sh, automataCI/ci.ps1 | *Execute* | The initialization script based on OS-specific platform, *.ps1* for Powershell; *.sh* for POSIX shell. |
| **1** | automataCI/[TYPE] | **Source** | Source the job related executables and run it during sourcing. |
| | automataCI/_[TYPE] | **Source** | Source the job related executables' subroutine function and run it. This layer is meant to simplify the long execution *automataCI/[TYPE]* file. |
| **2** | [TECH]/[CI]/[JOB] | **Source** | Source the tech-specific job recipe that facilitates customization. |
| **3** | automataCI/services/* | **Source** | Source the required function libraries' services. |
| **4** | automataCI/services/io/* | **Source** | Source primitive function libraries' service. |

As noted above, understand that everything happens in AutomataCI is using *source* direction with the only exception to *ci.cmd* Polygot script initialization. Hence, unless you're working on *ci.cmd* and its sub-layers (*ci.sh* and *ci.ps1*) files, you shall **only use *return* as the exit command, not *exit***.

AutomataCI's Level 1 layers are expected to tell the journey of the CI execution pipeline with 100% pinpoint accuracy working alongside Level 2. **Both Level 1 and Level 2 are strongly encouraged to import and use Level 3 and Level 4 libraries for consistency and future-proof themselves** from low-level OS side changes.

It's duly noted that terminal printout (e.g. **stdout** & **stderr**) is strictly Level 1 layers role only. Other layers are forbidden to do so.

#### 4.2.2.3.1 . Re-Importable Level 3 Libraries

Given sufficient experimentation during prototyping stage, you're strongly advised to create function libraries for AutomataCI's customers at all times regardless how simple it is. **The important point of doing so is to be able to translate between POSIX Shell and PowerShell**.

Libraries are organized based on the designed purposes (e.g. *archive*, *checksum*, *compilers*, *compress*, *crypto*, *publishers*, and *versioners*). Then, the function suites are defined into their independent source-able script like *archive/tar.sh* or *archive/tar.ps1*.

All functions inside the function suites should have the leading library name of its suite in UPPERCASE form. Example, for *archive/tar* services, its functions should have leading name like: *TAR::create_xz* in POSIX Shell or *TAR-Create-XZ* in PowerShell.

It is duly noted that all functions facilitated here shall only:

1. **return 0** – if working fine.
2. **return 10** – considered working fine but signify the upstream to cancel the execution.
3. **return non-0 apart from 10** – error occurred.

**DO NOT attempt to do any terminal printout**. That's only Level 1 roles and jobs.

Hence, should you find any difficulty with reporting a state, please break that function down further granular where Level 1 can call each of them and storytelling correctly.

The function can perform concurrency or parallelism internally but all timelines shall always be converged back at the end of the function, to report back to the Level 1 story-telling timeline flow. Any outputs or dumps during concurrency or parallelism shall be dumped externally as a log file instead.

Libraries shall be created in a re-importable manner so that any script can safely import the libraries at any given time. The functions shall be parsed again and again throughout a job but each parsing shall always yield the same function definition. **Hence, DO NOT leave any code traces in any library script**.

Apache 2.0 License notice shall always be made available in the library script for:

1. In case anyone attempt to extract it, the license stays effective; AND
2. In case anyone wants to steal and perform copyright theft, these notices act as a line of defense; AND
3. to track all contributors in the copyright authorship list.

POSIX shell should have the mandatory '*#!/bin/sh*' as its first line, before the license clause. The template of the file shall be as follows:

```
#!/bin/sh
# Copyright {{ CONTIRBUTED YEAR }}  {{ NAME }} <{{ EMAIL }}>
# Copyright {{ CONTIRBUTED YEAR }}  {{ NAME }} <{{ EMAIL }}>
# ...
#
# 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.

SUITE::function_name_one() {
        ...

        return 0
}

SUITE::function_name_two() {
        ...

        return 0
}

...
```

PowerShell:

```
# Copyright {{ CONTIRBUTED YEAR }}  {{ NAME }} <{{ EMAIL }}>
# Copyright {{ CONTIRBUTED YEAR }}  {{ NAME }} <{{ EMAIL }}>
# ...
#
# 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.

function SUITE-Function-Name-One() {
        ...

        return 0
}

function SUITE-Function-Name-Two() {
        ...

        return 0
}

...
```

Both POSIX Shell and PowerShell scripts of the same suite stays in the same directory next to each other for keeping the import clause consistent.

#### 4.2.2.3.2 . The Primitive Level 4 IO libraries

Like Level 3 libraries with ONE exception: **ALL LEVEL 4 library suites MUST BE SELF-CONTAINED**. This means **any library suites falls in *automataCI/services/io* directory are primitive building blocks and shall not import anything including each others**.

### *4.2.2.4 . Keep Everything in AutomataCI Directory*

To ensure future upgrades availability and ease of distributions, please keep every AutomataCI stuffs inside *automataCI/* directory. As demonstrated in the quick start section, AutomataCI customers shall only update *automataCI/* directory from the upstream and perform any localized patching later.

We only want to distribute that directory and that's it.

## 4.3 . Upstream Process

This section covers how to upstream your AutomataCI's contributions to the upstream distributions. They are described in details for the following sub-sections.

### 4.3.1 . Raise an Issue Ticket in Forum

Start-off by raising an issue ticket in the designated forum (usually GitHub) and discuss about it. You will be surprised how much this helps to expedite problem solving and sometimes time, don't have to do much at all.

### 4.3.2 . Clone the AutomataCI repository and Perform Development

Please clone the original AutomataCI repository and develop there. The upstream maintainers only accepts AutomataCI upstream repository only and shall reject any patches not from those repository.

### 4.3.3 . Set Patches / Pull Request, Code Review

Once done, please make it available for the maintainers to code review your changes. You can git format patch your commits and upload it into your issue ticket raised earlier.

### 4.3.4 . Acceptance

There are 2 possible acceptance outcomes depending on how you approach the maintainers:

1. If you're sent in by patch, the maintainer shall apply it to the repo and signed it using his/her GPG key.
2. If you're sent in by pull request, the maintainer shall perform merging locally and signed it using his/her GPG key, accepting your pull request to the upstream and then force-push to override the unsigned commits.

At this point, you're done for your part. Thank you.

# 5 . Release Strategy

AutomataCI uses Semantic Versioning system (https://semver.org/) to manage its release versions. To ensure sorting compatibility, a *v-* prefix is added to each version number when published.

Unlike other strategies, AutomataCI approach forward increment aggressively by only supporting the latest version of itself expiring all supports to the previous versions. However, that does not means the customers have to immediately upgrade it to the latest and greatest due to AutomataCI's product nature.

In special (and rare) cases, AutomataCI may deploy release candidate (*~rcN*) strategy to perform external ecosystem testing where:

1. N is candidate number; AND
2. **always uses tilde (~) and not dash (-)** to avoid conflicting with some mainline distributing ecosystems.

All release candidates **ARE NOT a latest release**.

# 6 . Epilogue

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