Automata Engineering Specification

Version 1.5 International English (en), September 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 us 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. Insert SECRETS.toml (OPTIONAL)

Please request for the *SECRETS.toml* containing secret-related data required for certain CI job. This file is gitignored by default for security reason.

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 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.

2.9 . 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.

2.10 . 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.

2.11 . 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.

2.12 . 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.

2.13 .   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*

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

2.14 .   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 creatded 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 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 using Homebrew; Windows 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 (for now)
*freebsd	FreeBSD OSes	freebsd
dragonfly*	Dragonfly OSes	dragonfly

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 requires confidentiality.

This file is entirely optional and 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.
build/	➡ default build output directory.
pkg/	➡ default package output directory.
resources/	➡ default indirect raw materials and assets housing directory.
resources/packages	➡ housing all packages control template files.
resources/icons	➡ housing all graphics and icon files.
resources/licenses	➡ housing all project licensing generative documents.
resources/docs	➡ housing all project's document generators.
resources/changelog	➡ housing all project's changelog entries data.
src/	➡ house common source codes (base directory).
src/.ci/	➡ house common source codes CI job recipes (applies to all).
src[TECH]/	➡ house tech-specific source codes (base directory).
src[TECH]/.ci/	➡ house tech-specific source codes CI job recipes.
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 same context but different aspect separations; AND
3. No space is allowed.

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

{PURPOSE}_{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 permits and as long as it makes sense. 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

3.6.2 . 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*)

3.6.2.1 . *Operating Parameters*

This job place it output at the following path:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/{brand}[-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 .   Python Programming Language

For Python programming language, when using the provided srcPYTHON/ directory, it's executing the pip packages installation command in your **{PYTHON_PATH_ROOT}/{PROJECT_PYTHON}** directory:

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

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 .   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.

3.6.5.2.1 .   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
```

3.6.6 .   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*)

3.6.6.1 . *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

3.6.6.2 . *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}/

3.6.6.3 .   *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.

3.6.6.3.1 .   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.

3.6.6.3.2 .   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 . Package

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

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

This job importing sequences is as follows, where the latter overwrites the former:
1. baseline job recipe (e.g. *src/.ci/test_unix-any.sh* via *PROJECT_PATH_SOURCE*); AND THEN
2. tech-specific job recipe (e.g. *srcPYTHON/.ci/test_unix-any.sh* via *PROJECT_PYTHON)*

It's highly recommended to keep all the package CI job algorithms inside the baseline job recipe only. Unless absolutely necessary like PyPi is for solely for Python technology, it shall then be parked in tech-specific job recipe. *The latter and later always overwrite the former so only add the overriding functions in the tech-specific job recipe*.

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:

$${PROJECT\_SKU}[*]\_{OS}\text{-}{ARCH}[.{EXTENSION}]$$

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.

3.6.7.1 .   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).

3.6.7.2 .   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}/

### *3.6.7.3 .	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.

### 3.6.7.3.1 .	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**

3.6.7.4 .   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.

3.6.7.4.1 .   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 (Untested)	Supported (Untested)

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.

At this point, it is relatively unknown whether Windows OS and Mac OSX is fully supported since AutomataCI deploys its own compiler.

3.6.7.4.2 .   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"

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

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

AutomataCI provides the ability to overrides any existing required files (see below). If these files are absent, AutomataCI shall generate one using its generator functions. Be warned that creating these required files can be a cumbersome effort (due to its steep technical debt).

Hence, it is recommended to just focus on constructing the package's data path and leave the rest of the required files to the AutomataCI generative function.

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

3.6.7.4.2.1 . 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.

3.6.7.4.2.2 . Maintainers' Scripts

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

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

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

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

3.6.7.4.3 . 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 }}
```

3.6.7.4.4 . 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.

3.6.7.4.5 . Collaborating With Automation – Project Description Data

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

${__PROJECT_PATH_RESOURCES}/packages/DESCRIPTION.txt

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

3.6.7.4.6 . Testing Packaged DEB's Health

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

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

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

3.6.7.5 .   *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.

3.6.7.5.1 .   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.

3.6.7.5.2 . 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/${PROJECT_SKU}/
install -m 644 copyright %%{buildroot}/usr/local/share/doc/${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"
```

3.6.7.5.3 .   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"
```

3.6.7.5.4 . 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}/packages/DESCRIPTION.txt** resource file.

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

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

3.6.7.5.5 .   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.

3.6.7.5.6 .   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/

3.6.7.5.7 .   Collaborating With Automation – Project Description Data

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

${__PROJECT_PATH_RESOURCES}/packages/DESCRIPTION.txt

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

3.6.7.5.8 .   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.

3.6.7.5.9 . Testing Packaged RPM's Health

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

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

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

3.6.7.6 .   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

3.6.7.6.1 .   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**

3.6.7.6.2 . 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.

3.6.7.6.2.1 . 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

3.6.7.6.3 . 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**".

3.6.7.6.4 . 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.

3.6.7.6.5 . 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.

3.6.7.6.6 . 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.

3.6.7.6.7 . 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
```

3.6.7.6.8 . 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.

3.6.7.6.9 . 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.

3.6.7.7 .   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.

3.6.7.7.1 .   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

3.6.7.7.2 . 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.

3.6.7.7.3 . 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**.

3.6.7.7.4 . Testing Package

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

<div align="center">twine check "${__directory}/dist/"*</div>

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**

3.6.7.8 .   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.

3.6.7.8.1 .   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)

3.6.7.8.2 .   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.

3.6.7.8.3 . 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.

3.6.7.8.4 . 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

        ...
}
```

3.6.7.8.5 . 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.
DOCKER_USERNAME	SECRETS.toml	Used for Docker Login function.
DOCKER_PASSWORD	SECRETS.toml	Used for Docker Login function.

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

3.6.8.1 . 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**.

3.6.8.2 . 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.

3.6.8.3 .   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.

3.6.8.4 .   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}/

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

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

3.6.8.6 .   Debian Package (.deb)

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

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

The destination is set to:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/deb

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

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

3.6.8.6.1 .   Reprepro Database

AutomataCI 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_LOG}/reprepro

They're not set for reusable and shall be "purged and re-created" alongside the repository creation.

3.6.8.7 . 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.

3.6.8.8 . Red Hat Flatpak (Flatpak)

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

3.6.8.9 .   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**'.

3.6.8.9.1 .   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.

3.6.8.9.2 .   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/

3.6.8.9.3 .   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.

3.6.8.10 . 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.

3.6.8.11 . 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.

3.6.9 . Clean

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

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*)

3.6.10 . 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 similar product 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 .   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 so many. Here's an example:

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

versus

```
# POWERSHELL TYPE 1
$__output = Invoke-Expression "some_command `"…`""
if ($LASTEXITCODE -ne 0) {
    exit 1
}

# POWERSHELL TYPE 2
$__process = Start-Process `
        -Wait `
        -FilePath "$__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() {
        ...
}
```

<div align="center">versus</div>

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

The worst of all – Powershell's *return* is not behaving as commonyl 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 (competeable?) full-fletch programming language but lack of architecture planning is really bizzare.

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 like AI performances.

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 All Executions

To make sure the executions are re-usable, 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:

The function should comply to the following pattern:

```
# 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.
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.

REMEMBER: **Please make sure Powershell only uses positional parameters**. POSIX shell does not have any alternative so **stick to it.**

4.2.2.1.1 .   Output Data Processing Function

For output data processing function (e.g. lowcasing 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*, *$_target*)
   1.4 **Level 2 lowercase** – indicate private and disposable entity (e.g. *__run_this_internal_function*, *__Run-This-Internal-Function*, *$__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 to "generate a separate HTML documents and etc" natively*. Hence, **write in a way that folks will read the source code only**.

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 limit**.

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	Triggers	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 difficulty with reporting a state, 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 Leve 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. Epilogue

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