(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

Project Developer Guide

Version 1.0 International English (en), January 2023


(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

Table of Contents

(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

1 . Prologue

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

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

2 . Developer's Quick-Start

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

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

The steps are prepared in sequences.

2.1 . 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.

2.2 . To Setup The Repository for Development

To quickly setup the project repository simply:

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ git clone <project_url>
$ cd <project>
$ . ci.cmd setup


# ON WINDOWS SYSTEM
$ git clone <project_url>
$ cd <project>
$ .\ci.cmd setup
```

Upon completion, your repository is ready for development.

2.3 . To Start A Development

To start the development after setting up, simply:

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd start

# ON WINDOWS SYSTEM
$ .\ci.cmd start      # and execute any on-screen instructions just in case it fails.
```

Upon completion, your terminal should be ready for development.

2.4 . To Execute The Test Cycle

To run a test cycle, simply:

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd test

# ON WINDOWS SYSTEM
$ .\ci.cmd test
```

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

2.5 . To Build The Products

To build the production-ready product, simply:

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd build

# ON WINDOWS SYSTEM
$ .\ci.cmd build
```

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

2.6 . To Package Products

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

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd package

# ON WINDOWS SYSTEM
$ .\ci.cmd package
```

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

2.7 . To Release The Packages

To release the product, simply execute the following locally:

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd release

# ON WINDOWS SYSTEM
$ .\ci.cmd release
```

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

2.8 . To Compose All Documents

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

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd compose

# ON WINDOWS SYSTEM
$ .\ci.cmd compose
```

Upon completion, TODO: pending development.

2.9 . To Publish the Composed Documents

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

```
# ON UNIX SYSTEM (e.g. Linux & MacOS)
$ . ci.cmd publish

# ON WINDOWS SYSTEM
$ .\ci.cmd publish
```

Upon completion, TODO: pending development.

3 . Granted Licenses

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

ELABORATE IN DETAILS HERE

3.1 . AutomataCI Source Codes

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

(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

4 . Engineering Specification

The product engineering specification are defined here.

5 . Infrastructure – AutomataCI

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

5.1 . Why Another Automation CI

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

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

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

5.2 . The AutomataCI Mantra

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

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

5.3 . Technological Requirements

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

Generally speaking, you only need the following:

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

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

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

5.3.1 . External CI Services

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

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

5.4 . Filesystem

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

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

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

automataCI/	➞ house the projects' CI automation scripts.
automataCI/services	➞ house tested and pre-built CI automation functions.
bin/	➞ default build output directory.
pkg/	➞ default package output directory.
resources/	➞ housing all indirect raw materials and assets.
src/	➞ house actual source codes (base directory).
src/.ci/	➞ house source codes technology-specific CI job recipes.
tools/	➞ default tooling (e.g. programming language's bin/* executables).
tmp/	➞ default temporary workspace.
CONFIG.toml	➞ configure project's settings data for AutomataCI.
ci.cmd	➞ CI start point.

5.4.1 . File Extensions

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

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


(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

5.4.2 . Filename Convention

AutomataCI uses:

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

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

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

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

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

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

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

and place them next to each other will do.

5.5 . CI Jobs

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

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

Basically, the execution sequences are as follow:

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

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

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

5.5.1 . Setup

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

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

5.5.1.1 . Operating Parameters

This job place it output at the following path:

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

5.5.2 . Start

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

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

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

5.5.2.1 . Operating Parameters

This job takes its no special input path.

It generates output files and directories in the following path:

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

5.5.3 . Prepare

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

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

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

5.5.3.1 . Operating Parameters

This job takes the following as its input and output:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

AND/OR

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TEMP}/

5.5.4 . Test

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

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

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

5.5.4.1 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_SOURCE}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_LOG}/

5.5.5 . Build

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

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

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

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

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

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

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

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

5.5.5.1 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_SOURCE}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_BUILD}/

5.5.6 . Package

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

Since these ecosystem packaging 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.

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}[.exe]
```

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

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

5.5.6.1 . Cryptography Requirements

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

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

5.5.6.2 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_BUILD}/

{PROJECT_PATH_ROOT}/{PROJECT_PATH_TOOLS}/

This job generate files to the following as outputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}/

5.5.7 . Release

Release Job is responsible for publishing all the packages to their respective distribution ecosystems. It shall detect all the known packages in the `PROJECT_PATH_ROOT/PROJECT_PATH_PKG` (defined in `CONFIG.toml`).

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.

5.5.7.1 . Operating Parameters

This job takes the following as its inputs:

{PROJECT_PATH_ROOT}/{PROJECT_PATH_PKG}/

This job generate no files as output.

5.5.8 . Compose

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

TODO – under construction.

5.5.9 . Publish

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

TODO – under construction.

5.5.10 . Clean

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

TODO – under construction.

5.5.11 . Purge

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

TODO – under construction.


(Holloway) Chew, Kean Ho
hollowaykeanho@gmail.com | me@hollowaykeanho.com
www.hollowaykeanho.com

6 . Epilogue

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