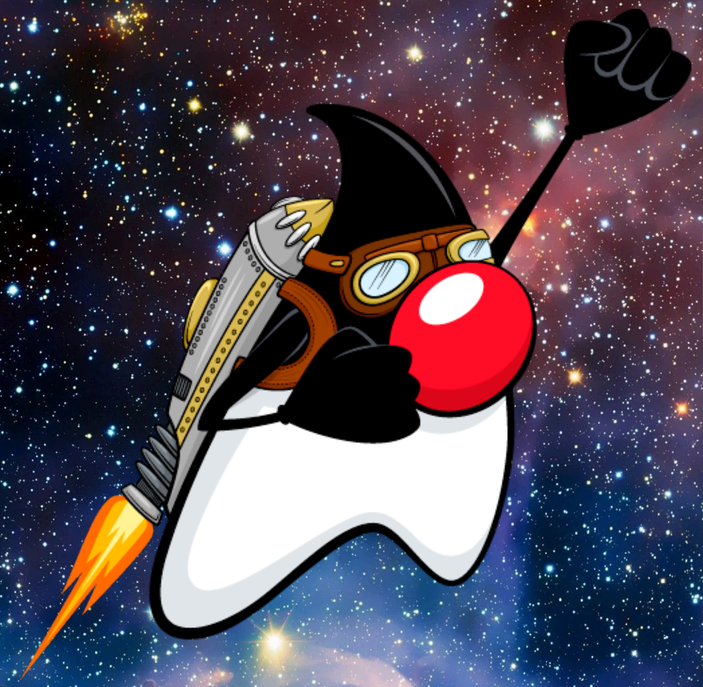
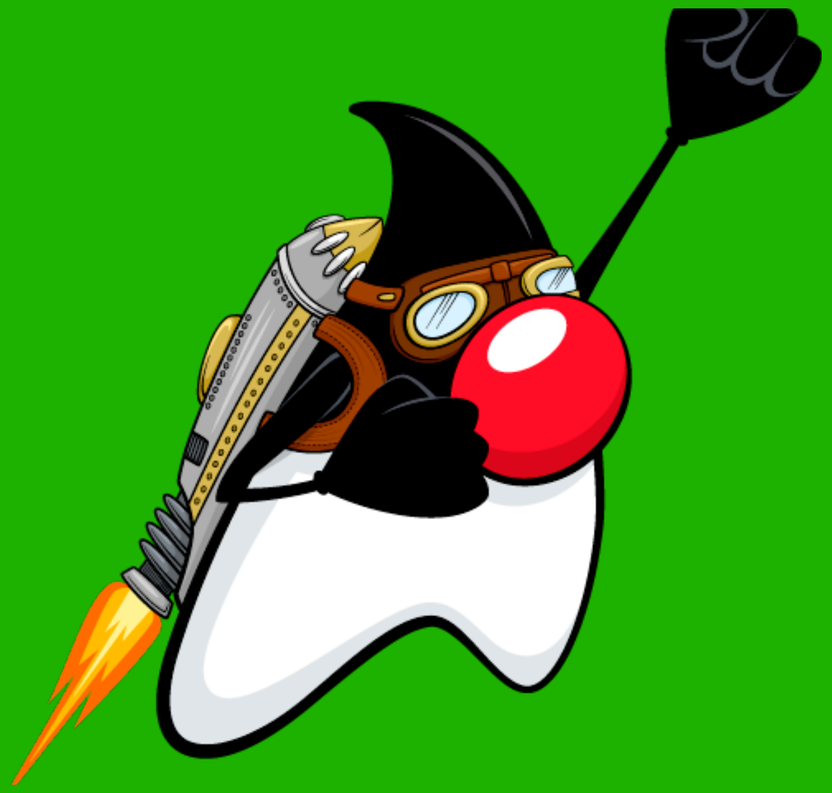


Do More with JReleaser

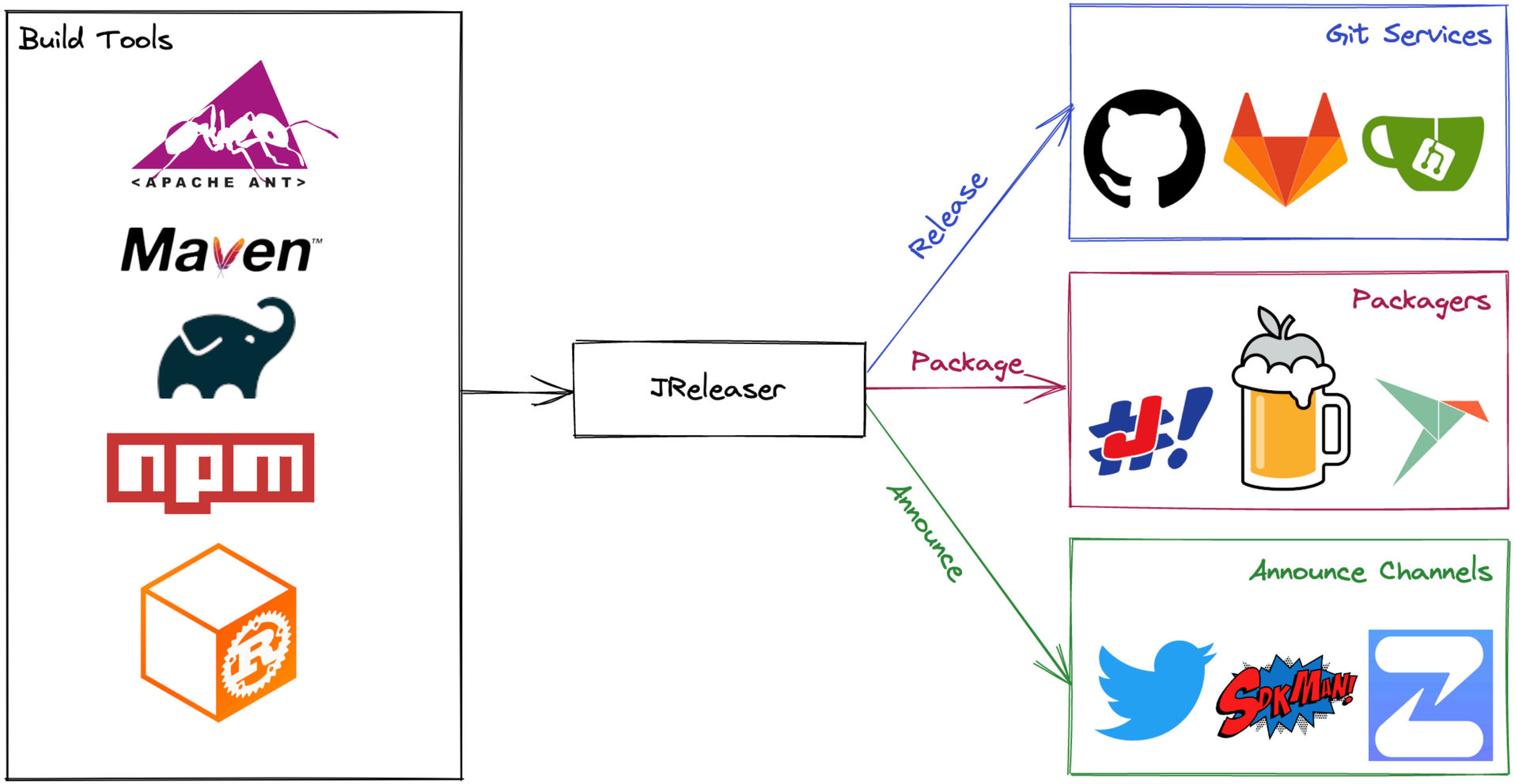


Introduction

What is it?

- Release Automation Tool
- Simplifies Chores
  - Tagging
  - Publishing Changelogs
- Publishes Artifacts
- Create Additional Binaries
- Inspired by Go's GoReleaser
- Works with various languages
- Additional Benefits for Java
- https://jreleaser.org

What it does



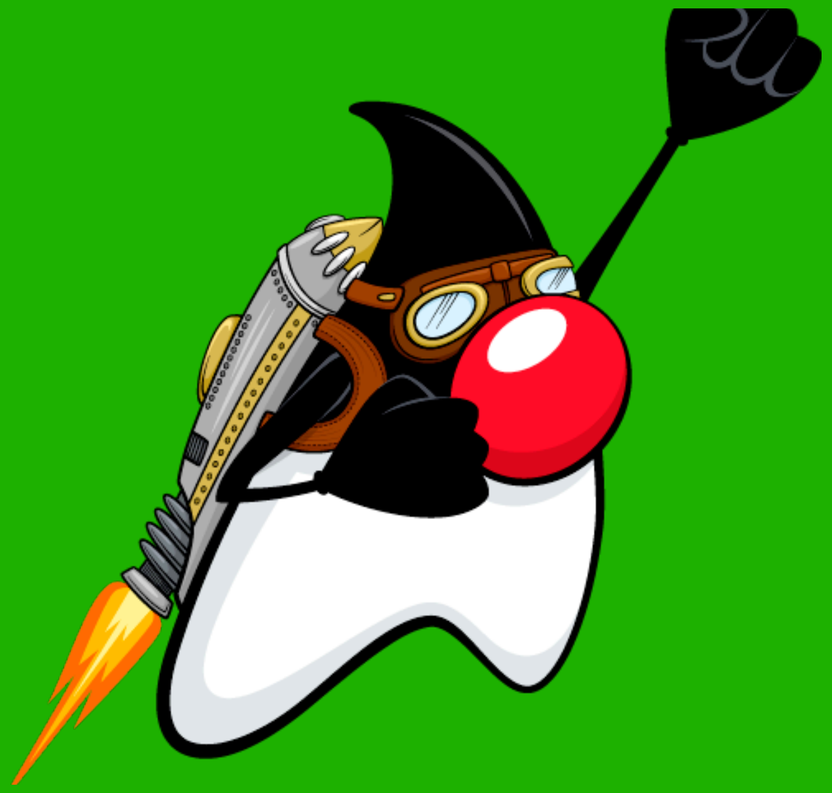
Source: https://jreleaser.org

How does it work?

- Create a release process in a `jreleaser[.yml|.toml|.json]` file
- You can also create release process in your favorite build tool
  - Maven, Gradle, Ant
- When ready you can run `full-release` in either the command line or part of your build tool
- All steps are configurable
- All steps are overridable from the command line for flexibility

Where does it run?

- Locally on your machine
- In a container
- In your favorite CI/CD tool
  - Many examples are available on the https://jreleaser.org website
  - Jenkins, Team City, Travis, Wercker, Circle CI, Github Actions, Gitlab CI



Installation

Installation

Sdkman

```
sdk install jreleaser
```

Homebrew

```
brew install jreleaser/tap/jreleaser
```

Curl Installation

Curl

```
// Get the jreleaser downloader
$ curl -sL https://git.io/get-jreleaser >
get_jreleaser.java

// Download JReleaser with version = <version>
// Change <version> to a tagged JReleaser release
// or leave it out to pull `latest`.
$ java get_jreleaser.java <version>

// Execute JReleaser
$ java -jar jreleaser-cli.jar <command> [<args>]
```

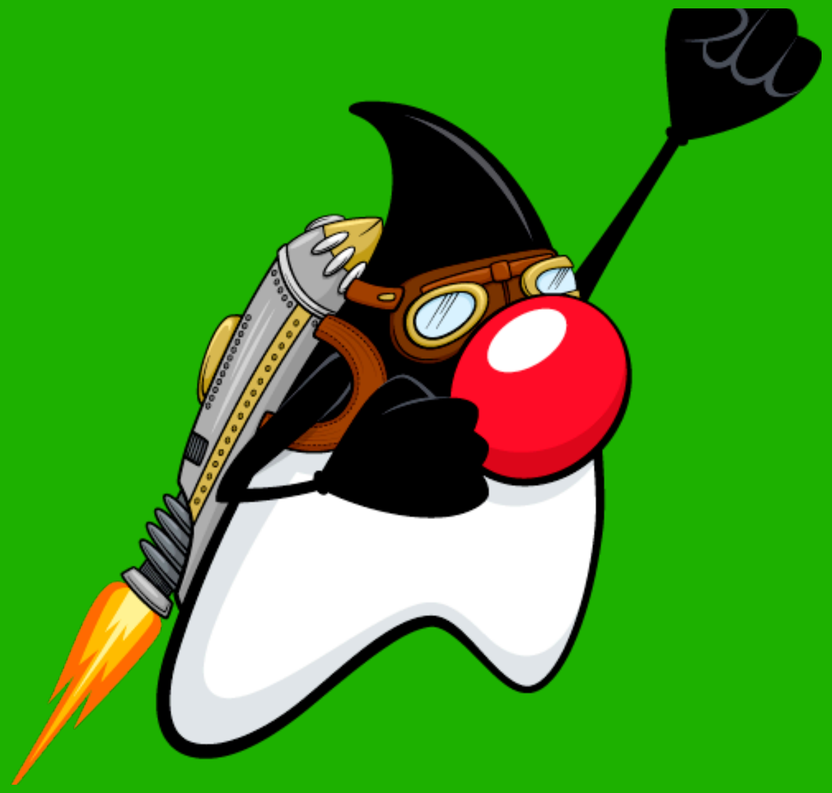
Docker

Docker

```
$ docker run -it --rm -v `(pwd)`:/workspace \
  jreleaser/<image>:<tag> <command> [<args>]
```

Docker Notes

- Images can be either `jreleaser-slim` or `jreleaser-alpine`
- The `jreleaser` command will automatically be run inside of `/workspace` folder
- Visit https://hub.docker.com/r/jreleaser/jreleaser-slim/tags for a list of docker tags



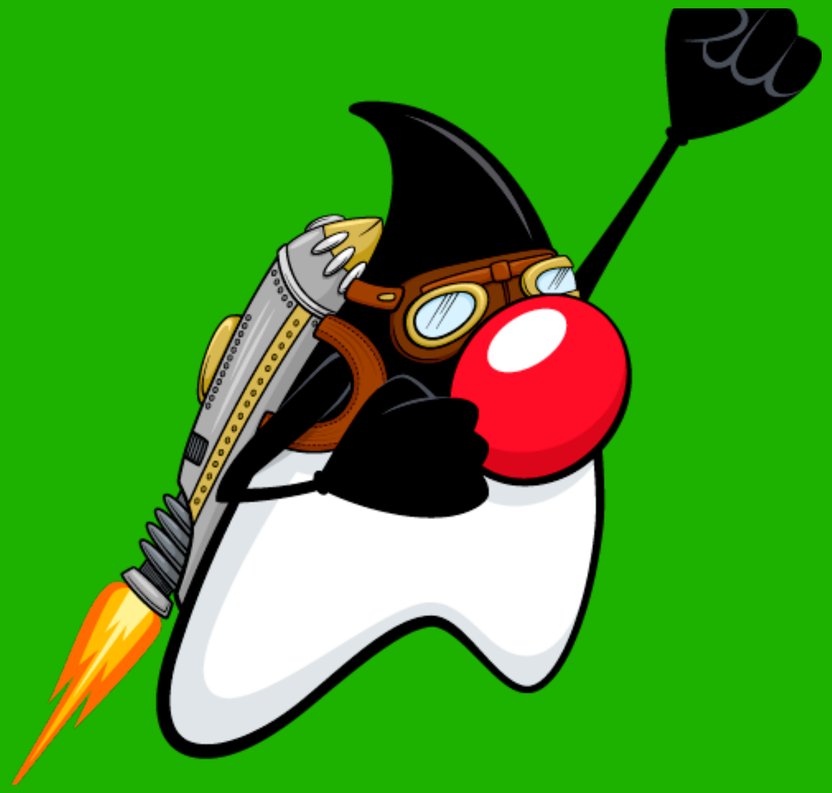
Starting

Command Line

Generates a JReleaser File

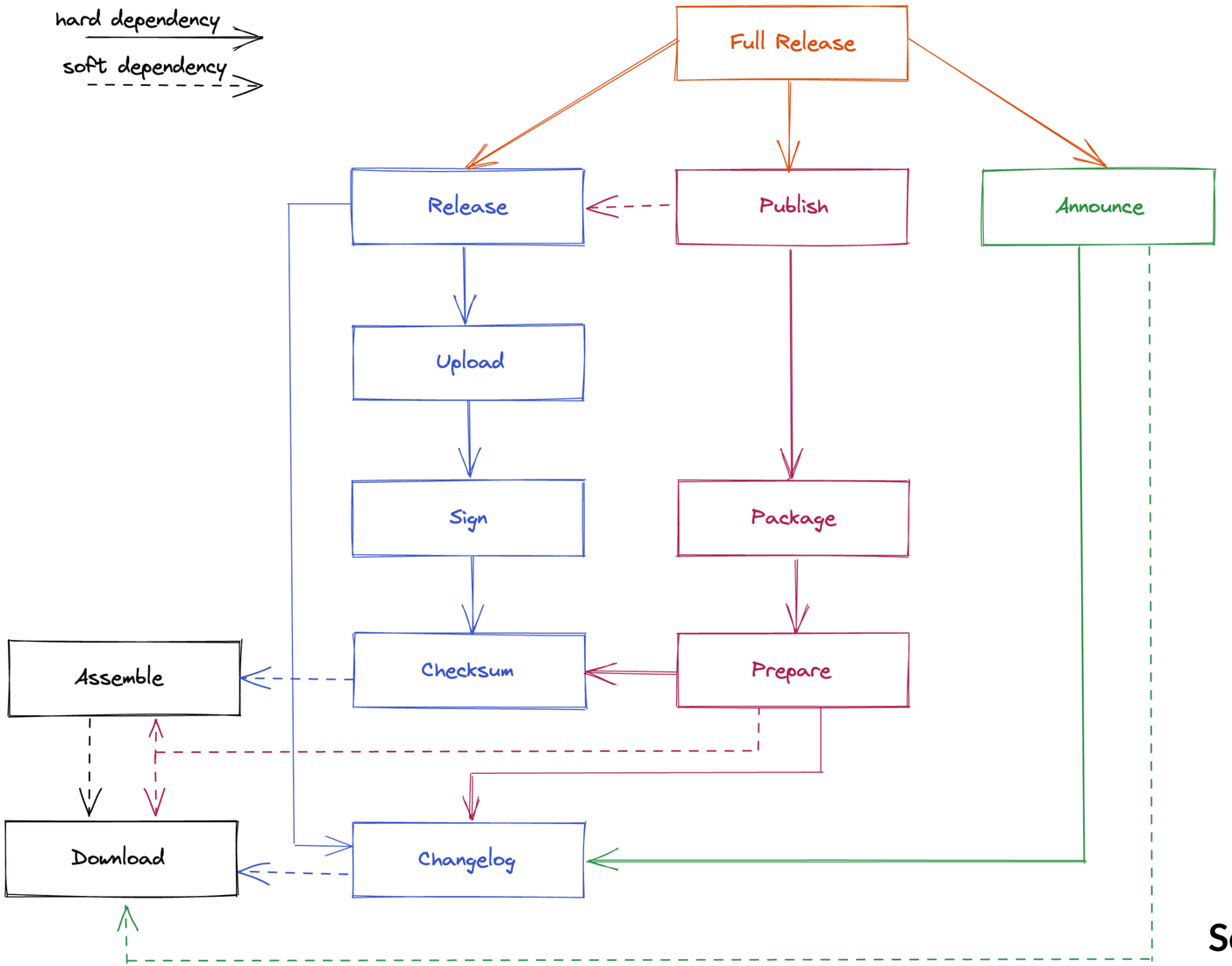
```
$ jreleaser init --format yml
[INFO]  Writing file /Home/duke/app/jreleaser.yml
[INFO]  JReleaser initialized at /Home/duke/app
```

JReleaser can create yml, toml, and json formats



Workflow

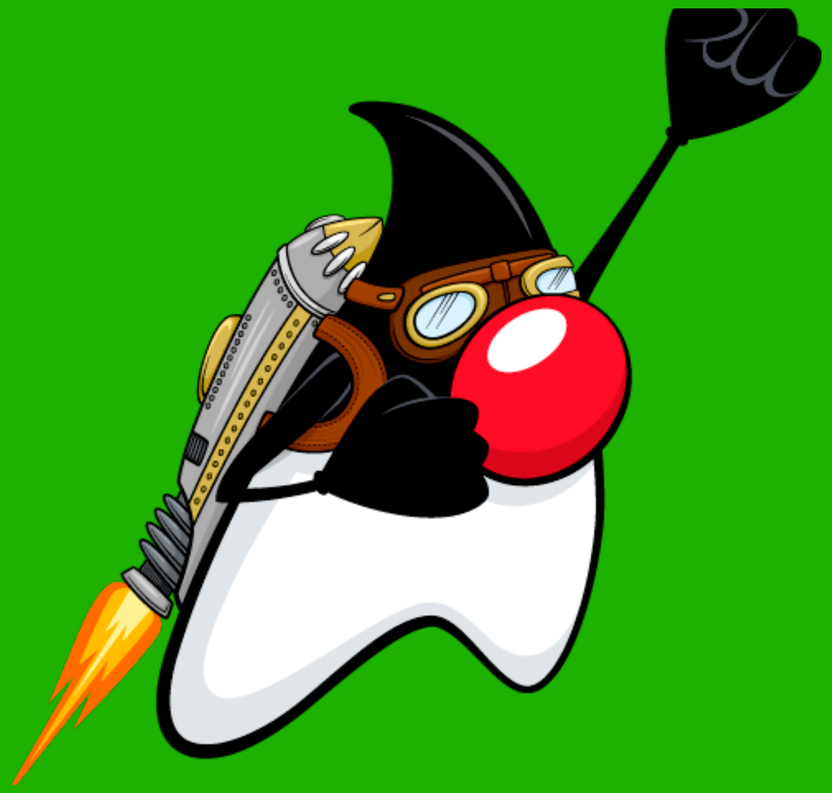
Workflow



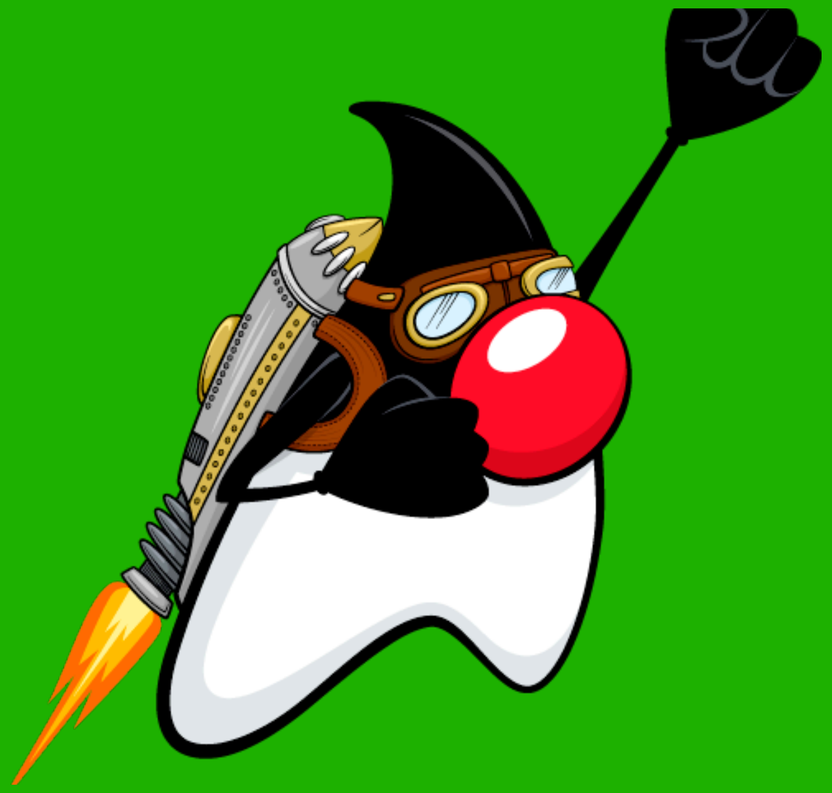
Source: https://jreleaser.org

Workflow Details

- *Full-Release* is the preferred step with JReleaser
- Each Stage is executed sequentially, if one fails, they all fail
- You may decide to resume from the start or from the last failed step
- Announcers are the exception, if they fail, build may continue, but a warning is printed



Demo: Running an Example in Maven, Docker, and Command Line



Configuration

Templates

Templating

- Fields in any configuration options can be parameterized using Mustache templates
- Mustache template take cue from C Templating using `{{placeholder}}`
  - The placeholder will be replaced/interpolated with actual value from JReleaser
- There are several defined *Name Templates* that can be used as placeholders in template files. These are common elements that you can use throughout your configuration
- Templates can be placed in `/src/jreleaser`

Mustache Templating

This is a common template. Notice that in_ca will be either true or false and will render based on its boolean result

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

Values

The values that will inevitably be interpolated

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 10000 - (10000 * 0.4),
  "in_ca": true
}
```

Interpolation Result

The result of the interpolation

```
Hello Chris
You have just won 10000 dollars!
Well, 6000.0 dollars, after taxes.
```

Mustache is logic-less

- *No Logic* mean there are no if statements, else clauses, or for loops
- There are only tags
- Some tags are replaced with a value, some nothing, and others a series of values

Tags

Tags

- Tags are indicated by the double mustaches {{}}
- {{person}} is a tag, as is {{#person}}
- Various tags available

Variables

- `{{name}}` - find the name key in the current context
- If there is no name key, the parent contexts will be checked recursively
- If the top context is reached and the name key is still not found, nothing will be rendered.
- All variables are HTML escaped by default, use `{{{name}}}` to use unescaped HTML

Mustache Misses and HTML

Given the following template, notice company comes in two flavors: double brackets and triple brackets

```
* {{name}}
* {{age}}
* {{company}}
* {{{company}}}
```

Values

The data, called the hash contains the following, notice that the company contains HTML

```
{
  "name": "Chris",
  "company": "<b>GitHub</b>"
}
```

Interpolation Result

The result of the interpolation

```
* Chris
*
* &lt;b&gt;GitHub&lt;/b&gt;
* <b>GitHub</b>
```

Sections

- Sections render blocks of text one or more times, depending on the value of the key in the current context.
- A section begins with a pound and ends with a slash
- `{{#person}}` begins a "person" section while `{{/person}}` ends it.

False Values/Empty Lists

If the person key exists, and has a value of false or empty list it will not be displayed

```
Shown.
{{#person}}
  Never shown!
{{/person}}
```

Values

Given the following values

```
{
  "person": false
}
```

Interpolation Result

It will result in the following

```
Shown.
```

Interpolation Result

It will result in the following

```
Shown.
```

Non-Empty Lists

If the person key exists and has a non-false value, the HTML between the pound and slash will be rendered and displayed one or more times.

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
```

Values

Given the following values

```
{
  "repo": [
    { "name": "resque" },
    { "name": "hub" },
    { "name": "rip" }
  ]
}
```

It will result in the following

```
<b>resque</b>
<b>hub</b>
<b>rip</b>
```

Lambdas

- When the value is a callable object, such as a function or lambda, the object will be invoked and passed the block of text.
- The text passed is the literal block, unrendered
- `{{tags}}` will not have been expanded - the lambda should do that on its own
- In this way you can implement filters or caching

```
{{#wrapped}}
  {{name}} is awesome.
{{/wrapped}}
```

Values

Given the following values

```
{
  "name": "Willy",
  "wrapped": function() {
    return function(text, render) {
      return "<b>" + render(text) + "</b>"
    }
  }
}
```

Interpolation Result

The result of the interpolation

```
<b>Willy is awesome.</b>
```

Non False Values

- When the value is non-false but not a list, it will be used as the context for a single rendering of the block.

Given the following template, notice that `person?` is a block

```
{{#person?}}
  Hi {{name}}!
{{/person?}}
```

Values

Given the following values

```
{
  "person?": { "name": "Jon" }
}
```

Interpolation Result

The result of the interpolation

```
Hi Jon!
```

Inverted Sections

- Inverted sections begins with a caret (hat) and ends with a slash.
- They will be rendered if the key doesn't exist, is false, or is an empty list.

Given the following template, notice that `^repo` is a block and will be rendered if empty or false

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
{{^repo}}
  No repos :(
{{/repo}}
```

Values

Given the following values

```
{
  "repo": []
}
```

Interpolation Result

The result of the interpolation

```
No repos :(
```

Comments

Comments are a tag with a ! and will be ignored

```
<h1>Today{{! ignore me }}.</h1>
```

Here is the result of the interpolation

```
<h1>Today{{! ignore me }}.</h1>
```

Files

Files

- You can define a set of additional files that should be uploaded as part of the release.
- These files may also be checksummed and signed before uploading.
- When not explicitly set, the value of `active` may be resolved from an environment variable `JRELEASER_FILES_ACTIVE` or from a system property `jreleaser.files.active`.
- The system property takes precedence over the environment variable

Environment

Environment

- Configure fields from external sources
- Allows keeping a stable configuration file and only update the external sources to produce a new release
- Values may be read from a Java properties file or from environment variables
- External Environment Variables has precedence over environment variables
- Configuration file is located at `~/.jreleaser/config.properties`
- The properties file should have no nesting
- Use toml or yaml for multiline configuration

Properties from Maven

- JReleaser will honor environments from Maven properties from:
  - The command line by using the `-D` flag.
  - `pom.xml` by using the `<properties>` block.
  - Settings file using the `<properties>` block inside an active profile.

Precedence for Maven

- JReleaser will adhere to the following precedence
  - Values defined in the model.
  - Values defined as Maven properties.
  - Values defined in `<releaser><environment><variables>`.
  - Environment variables
- Keys must be in uppercase separated by underscores or or fully lowercase words separated by dots

Properties from Gradle

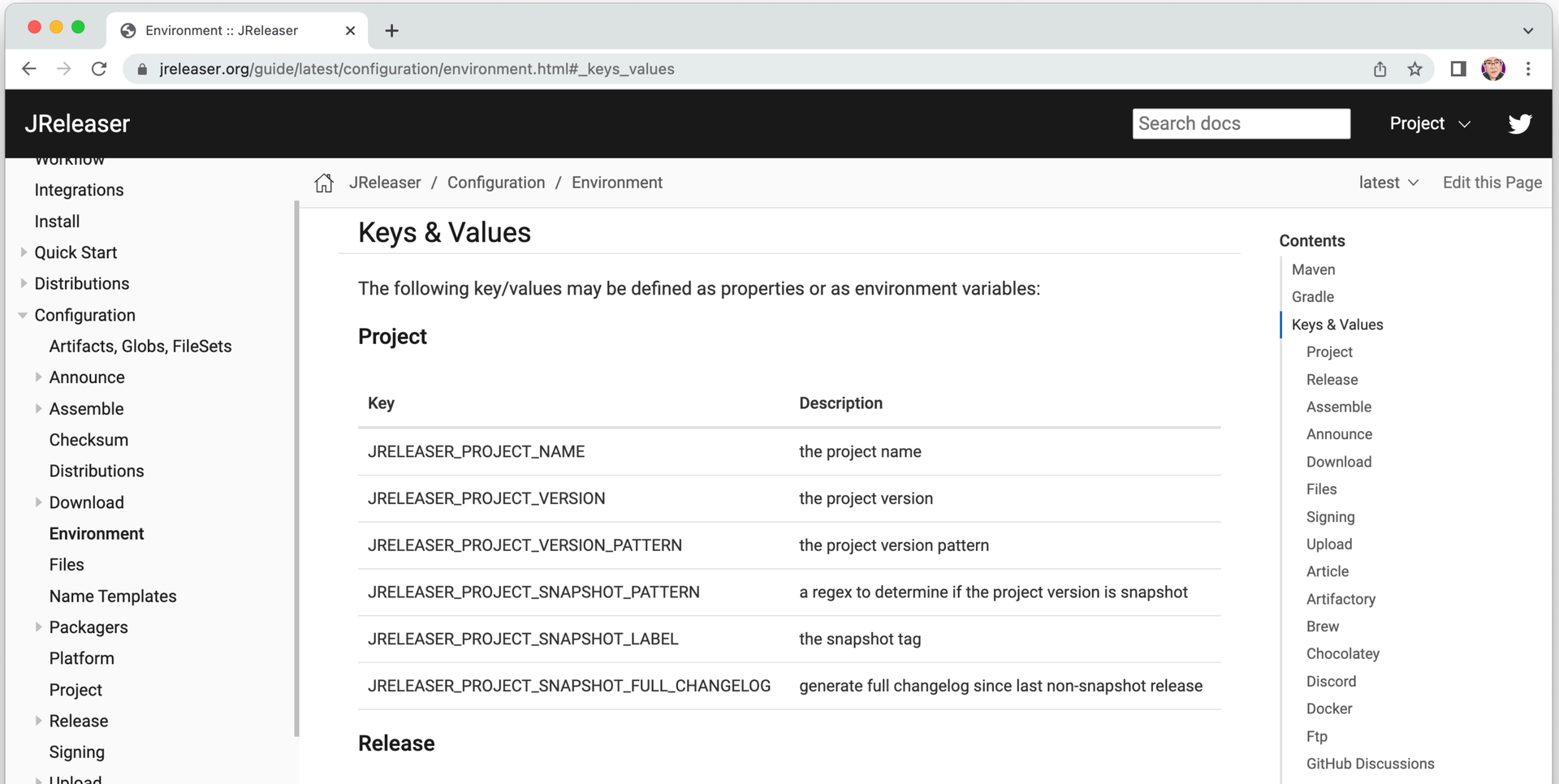
- JReleaser will honor environments from Gradle properties from:
  - The command line by using the `-P` flag.
  - `gradle.properties` in the adjacent to the project file
  - Global `gradle.properties` located in your home directory

Precedence for Gradle

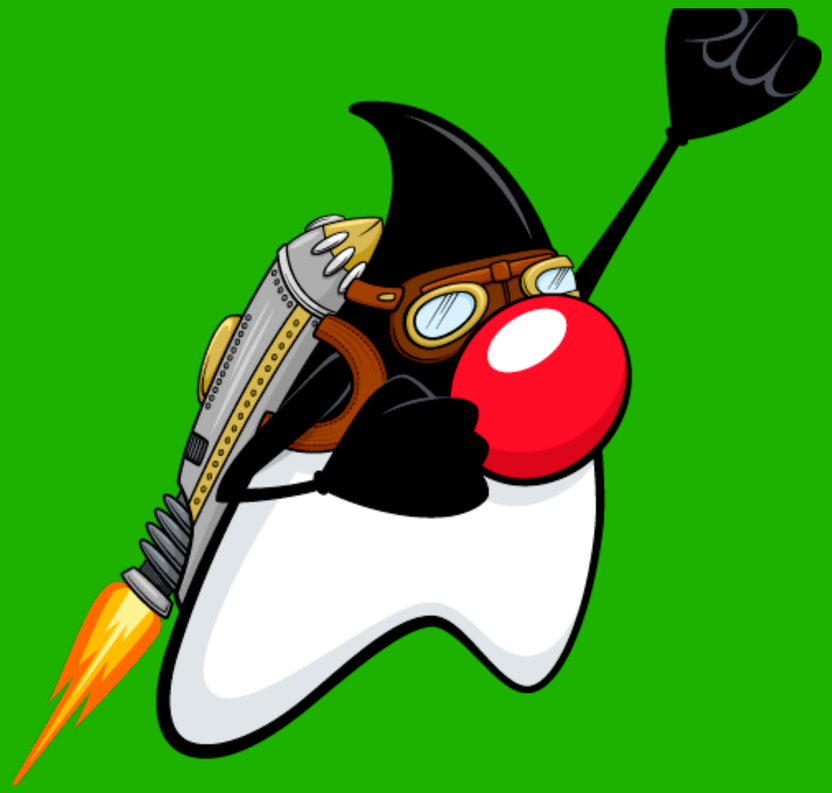
- JReleaser will adhere to the following precedence
  - Values defined in the model.
  - Values defined as Gradle project properties.
  - Values defined in `<releaser><environment><variables>`.
  - Environment variables
- Keys must be in uppercase separated by underscores or or fully lowercase words separated by dots

List of Key/Values

https://jreleaser.org/guide/latest/configuration/environment.html#_keys_values

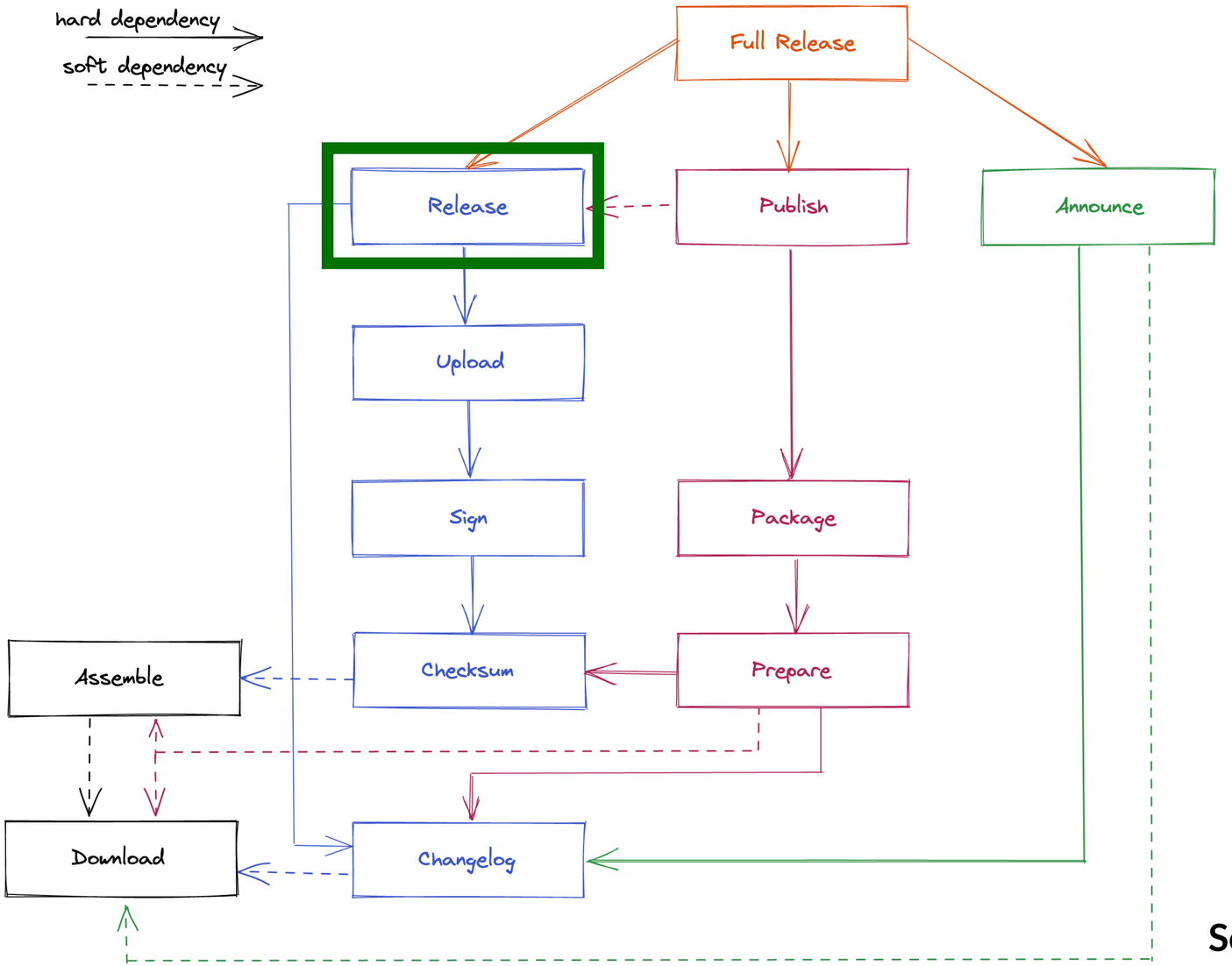


Built-in Variable & Functions



Release

Release Step



Source: https://jreleaser.org

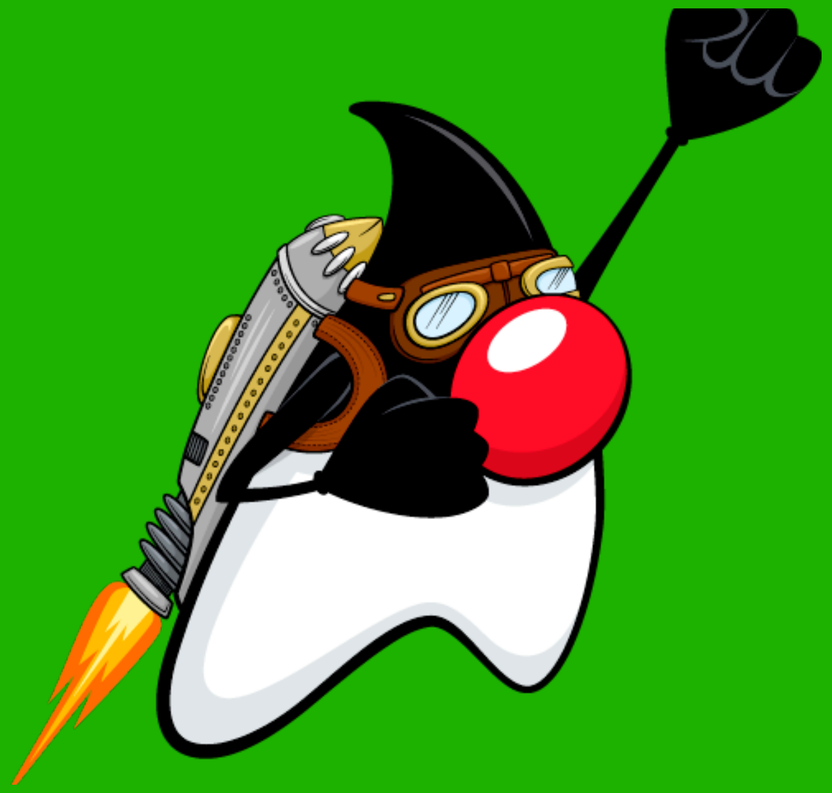
Releasing

- Create and release a tag
- Upload files to a destination
- Generate a changelog from new commits since the last tag
- If a matching milestone is found then it will be closed
- Only one single *release service* can be configured at a time

Release Services

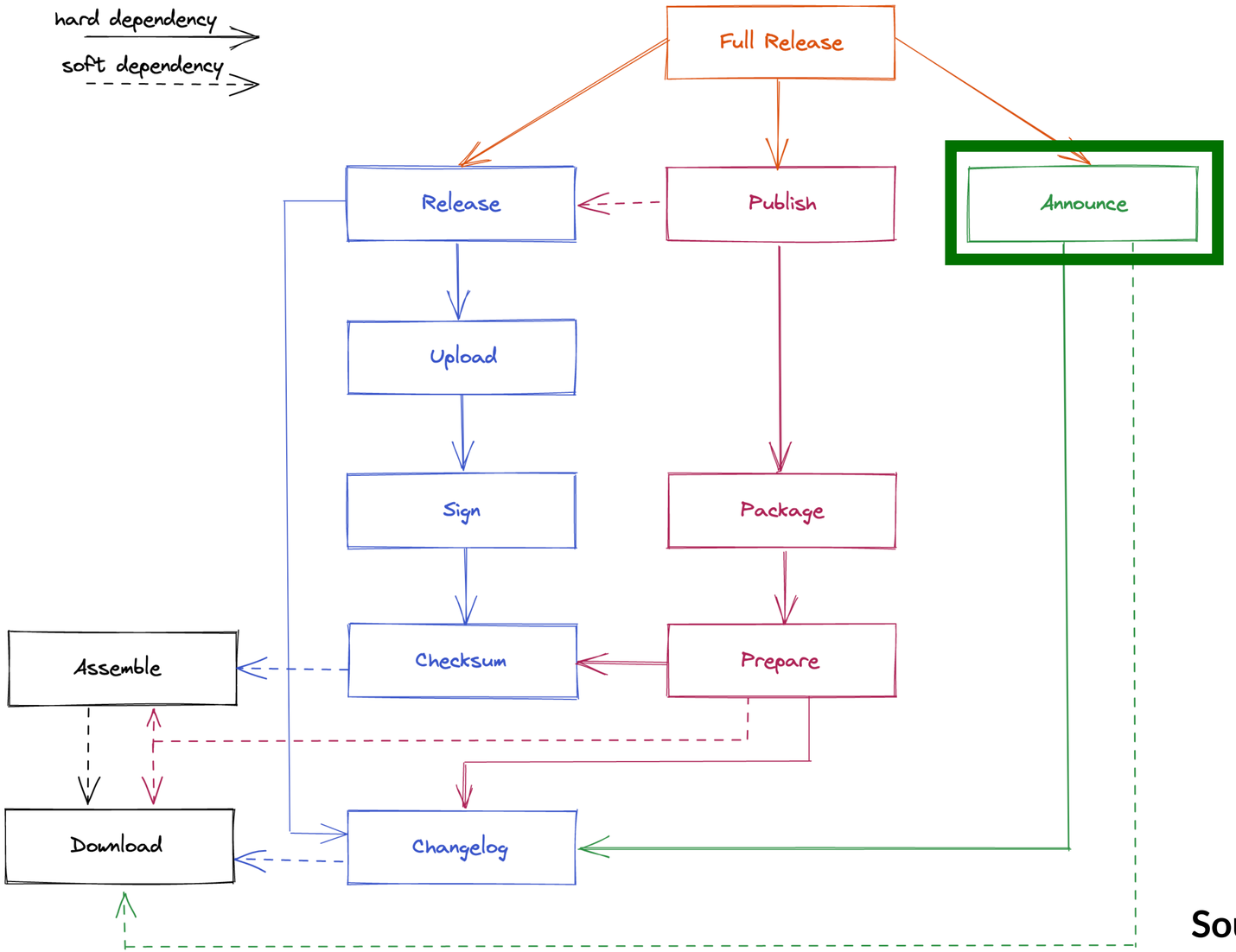
List of Release Services

- Codeberg
- Generic - If not specific releaser is available
- Gitea
- Github - Sent to the backend https://api.github.com
- Gitlab



Announce

Announce Step

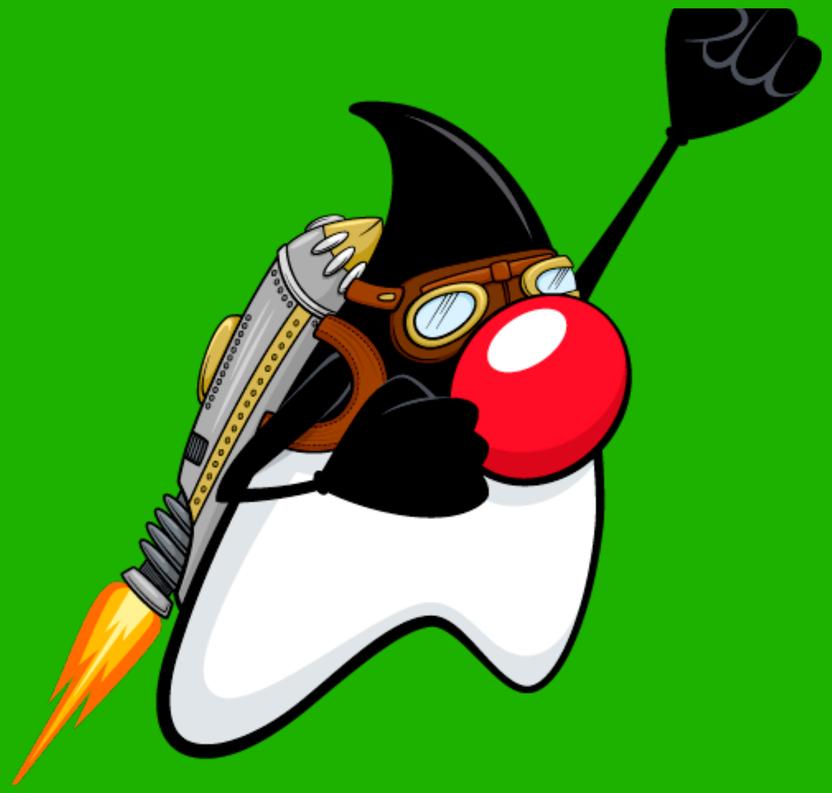


Source: https://jreleaser.org

Announce

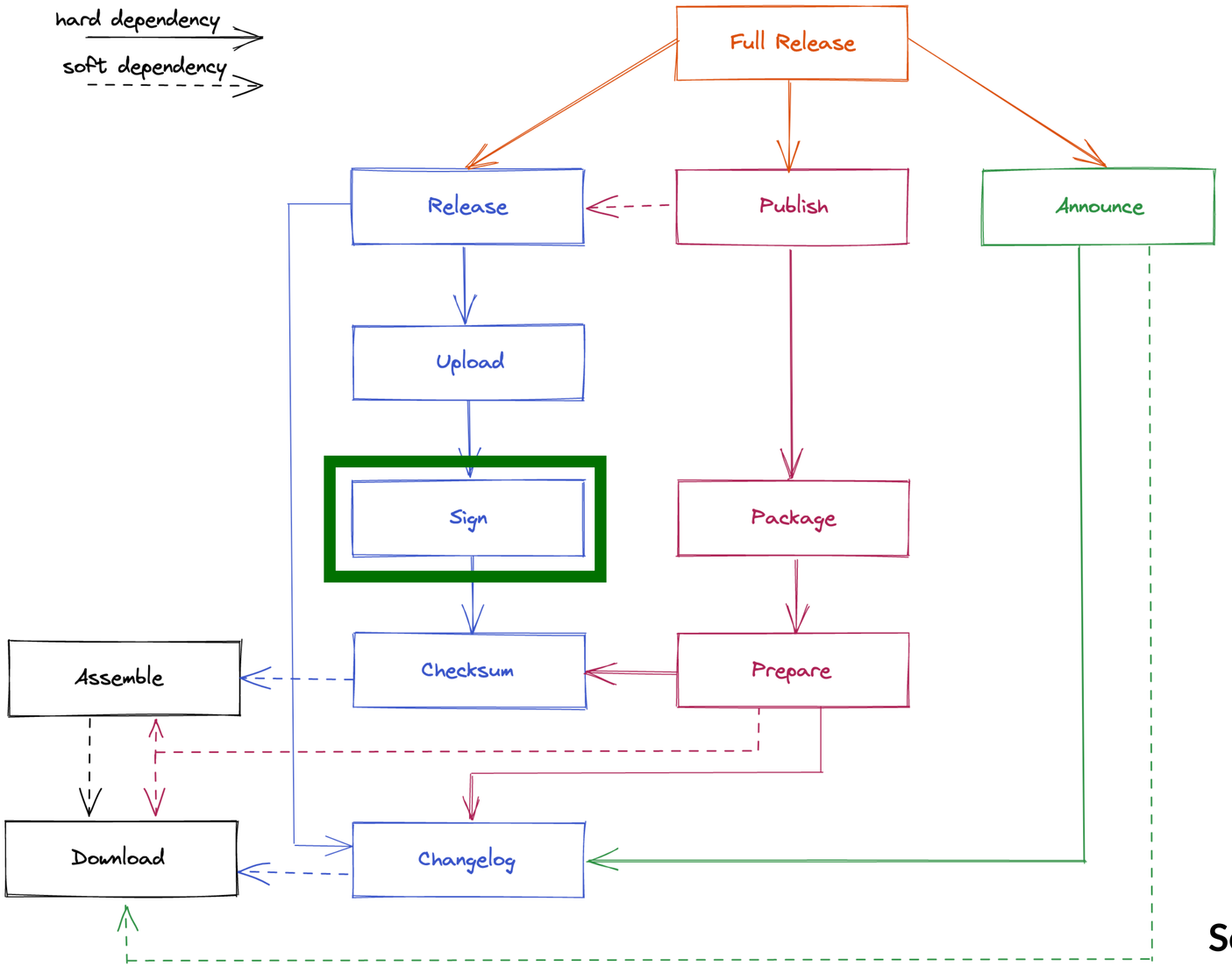
Announce has various ways to connect to your favorite way to announce brand new releases

- Article
- Discord
- Github Discussions
- Gitter
- Google Chat
- Mail
- Mastodon
- SDKman
- Slack
- Teams
- Telegram
- Twitter
- Webhooks
- Zulip



Signing

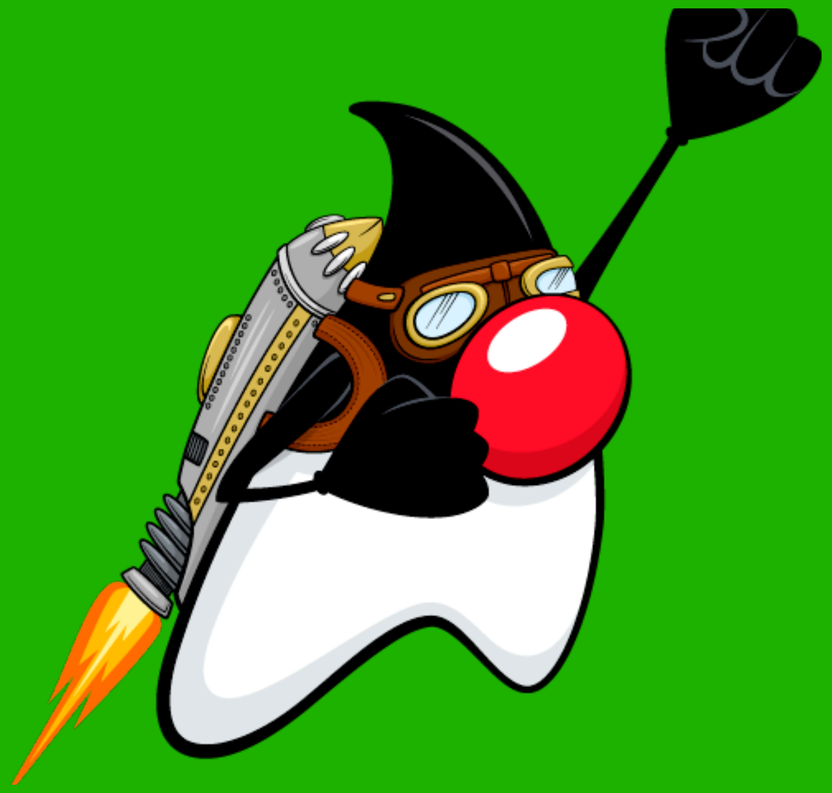
Signing Step



Source: https://jreleaser.org

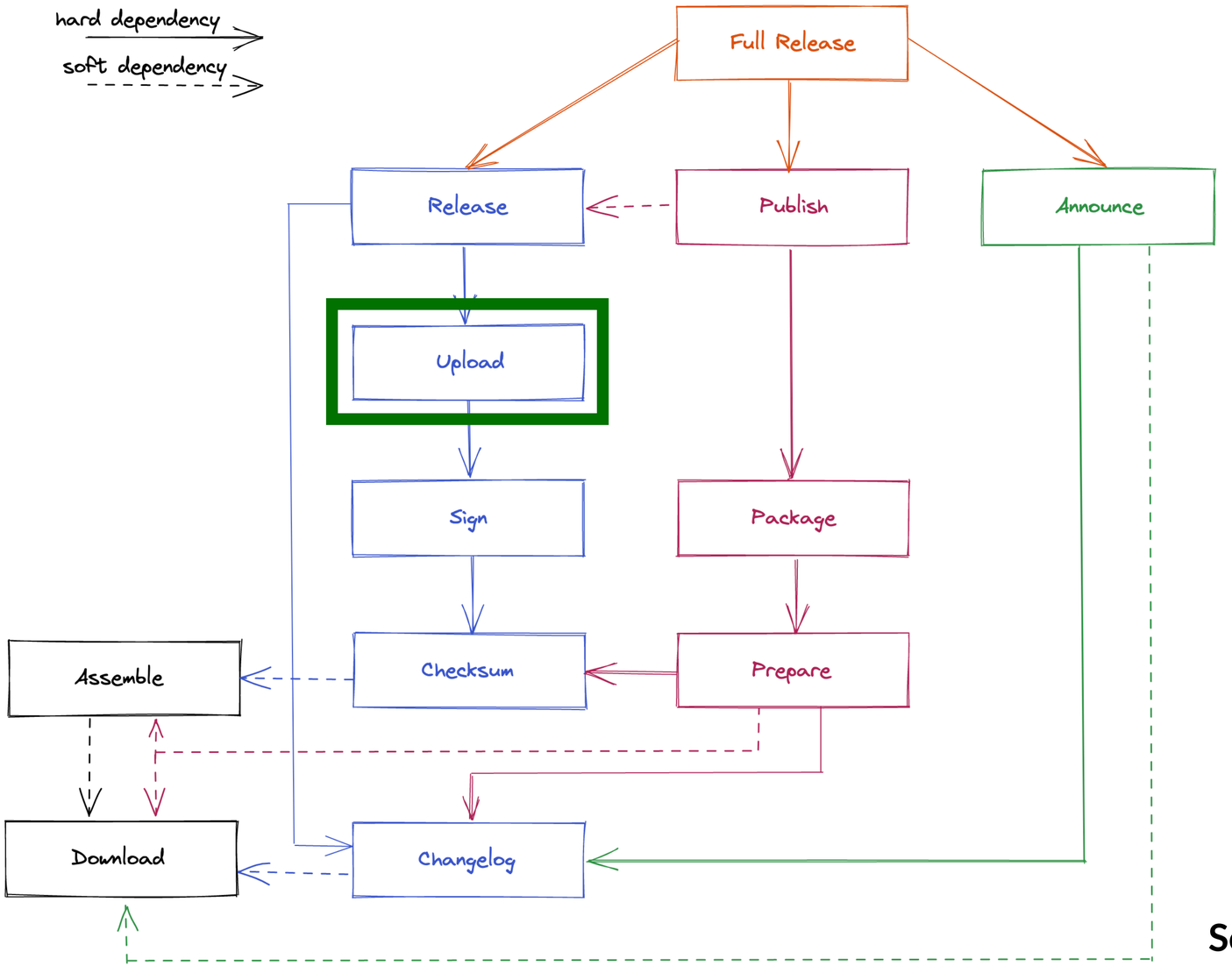
Signing

- Creates PGP signatures for all input files.
- This includes
  - All artifacts per distribution
  - Every matching file configured in the Files section
  - Output files from the Checksum step (“checksums.txt")
- This step is optional and can be disabled if needed



Upload

Upload Step



Source: https://jreleaser.org

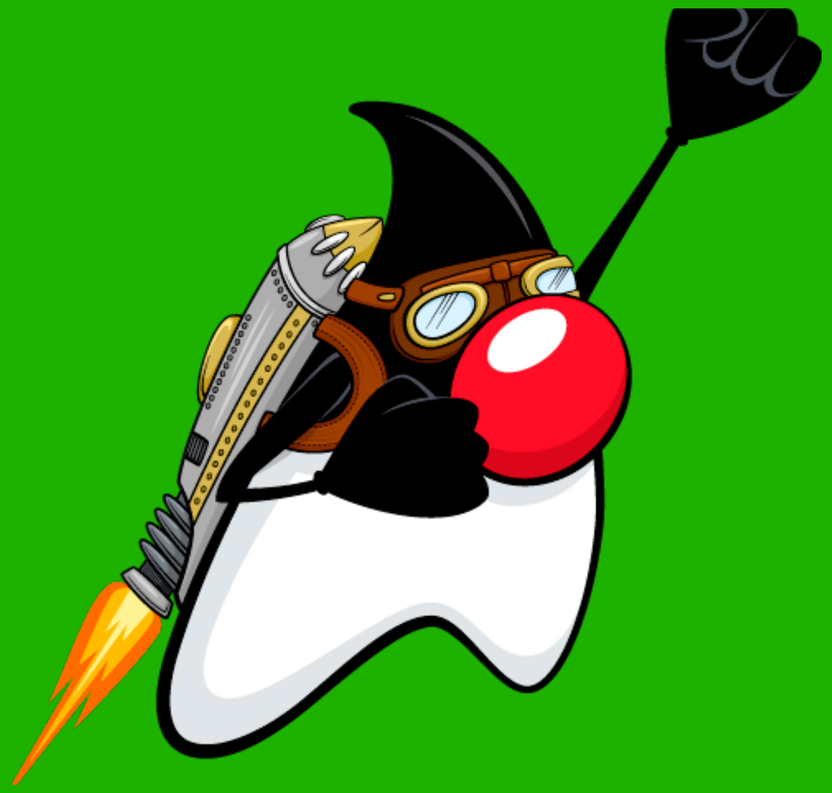
Upload

- Uploads artifacts and files to the configured destinations.
- This step uploads:
  - Signatures (if enabled)
  - All distribution artifacts
  - Every matching file configured in the Files section.
- Depends on the *Signing* step

Upload

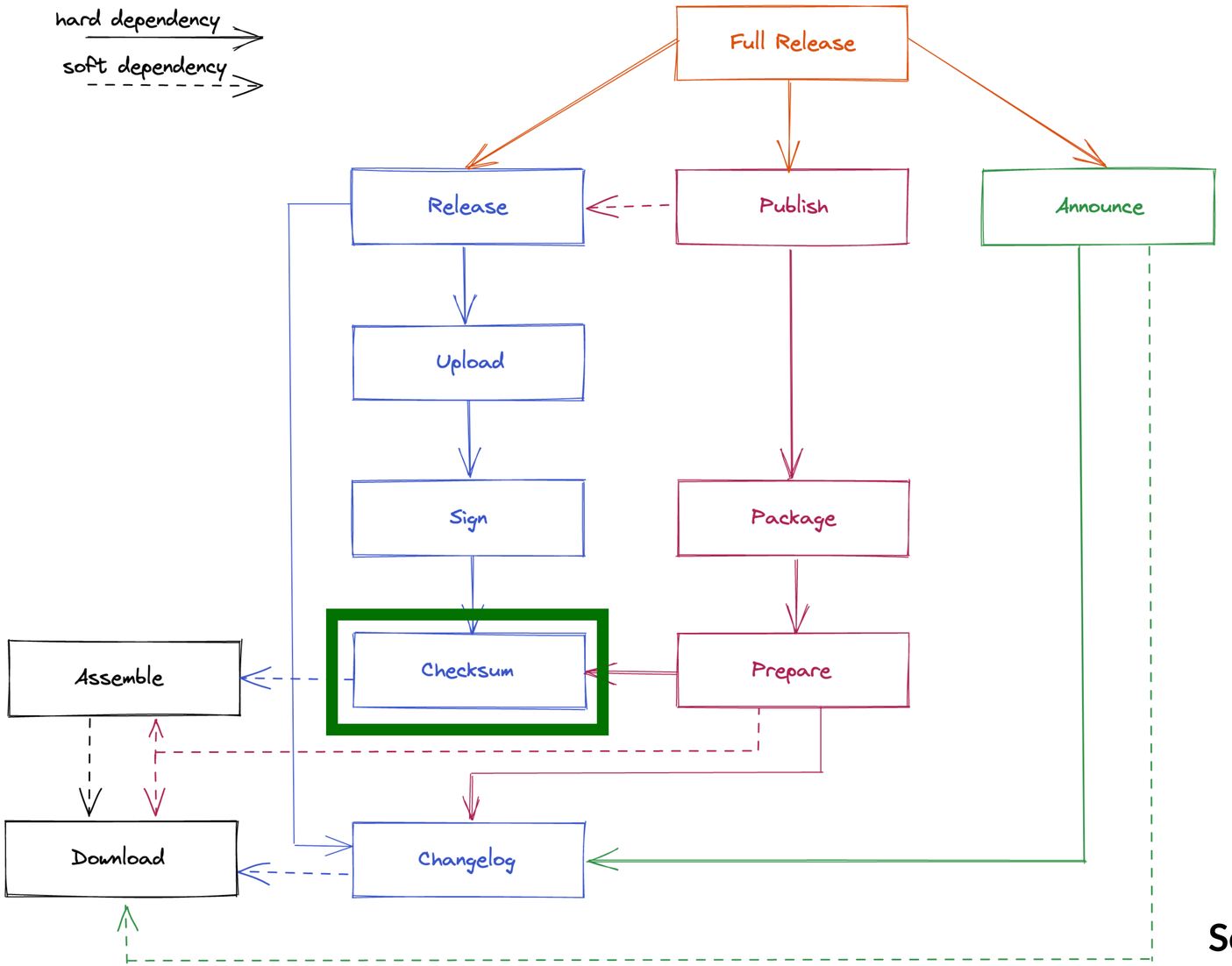
Artifacts and files may be uploaded to configured locations.

- Artifactory
- FTP
- HTTP
- AWS S3
- SCP
- SFTP



Checksums

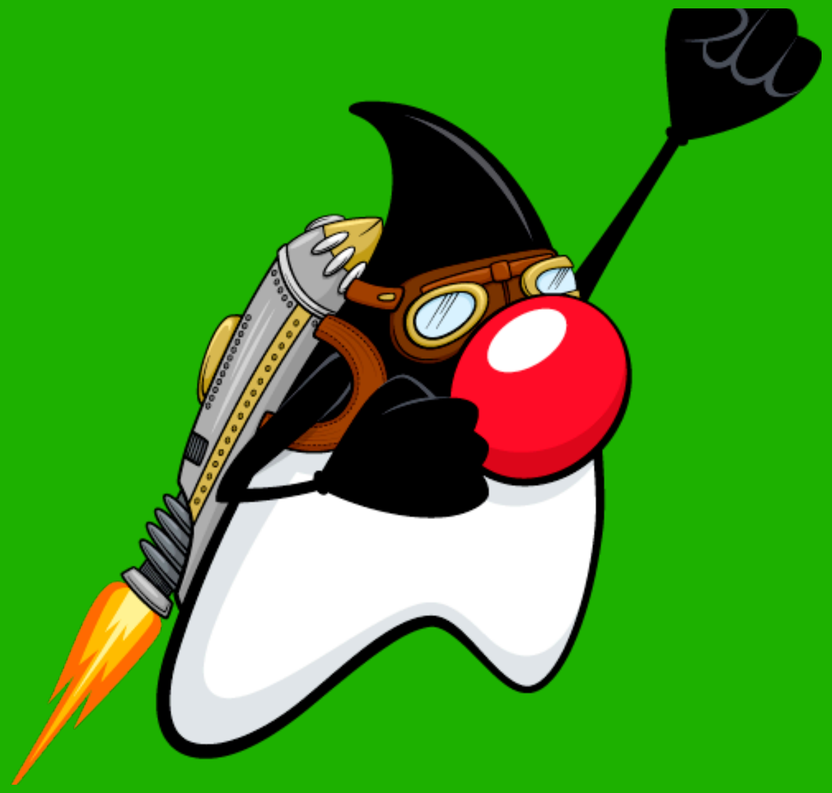
Checksum Step



Source: https://jreleaser.org

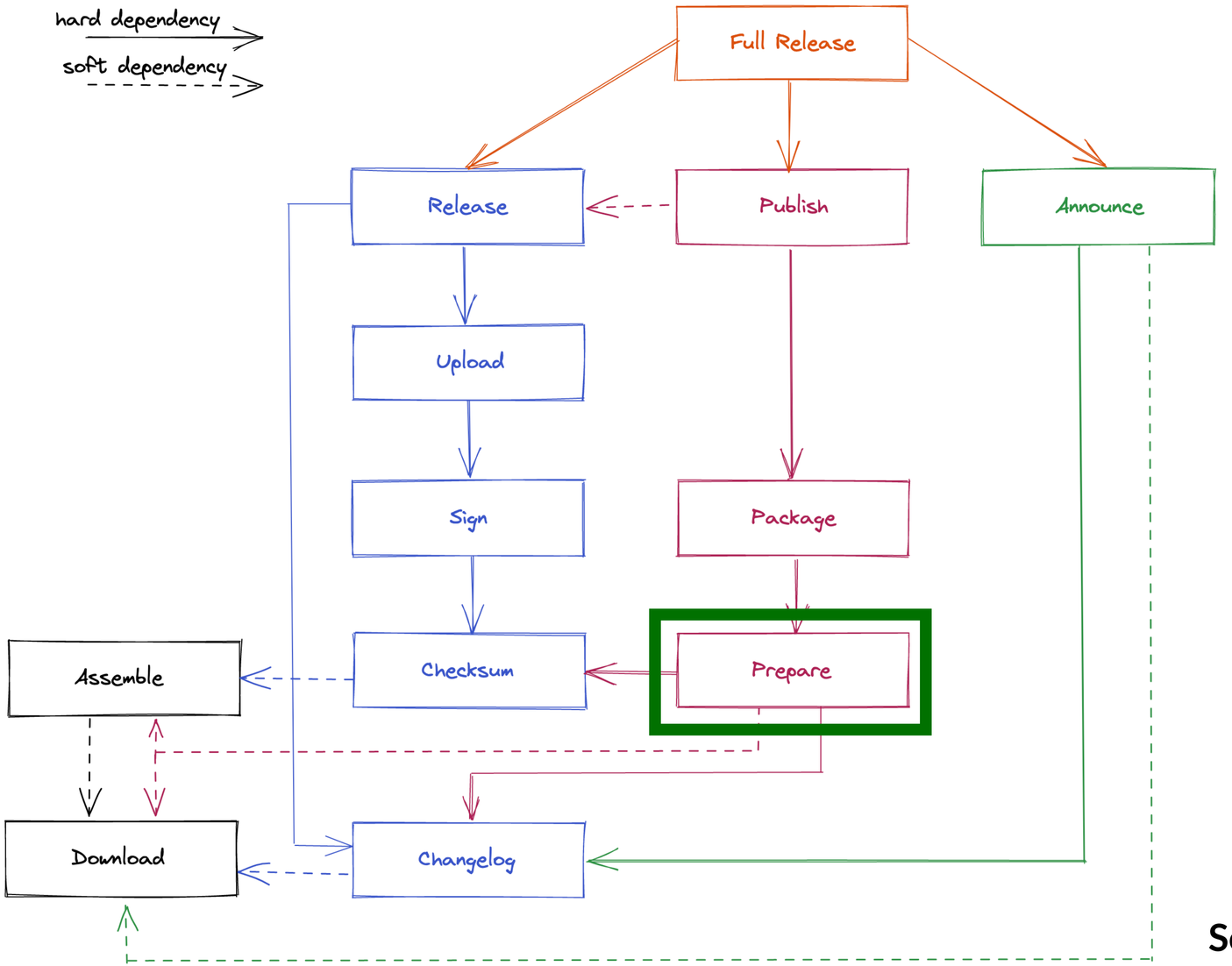
Checksum

- Calculates SHA256 checksums (as well as other algorithms) on all input files.
  - This includes all artifacts per distribution
  - Every matching file configured in the `Files` section
- All files must exist by the time Checksum is invoked, otherwise an error will occur
- Checksums will be placed at `${basedir}/out/jreleaser/checksums`



Prepare

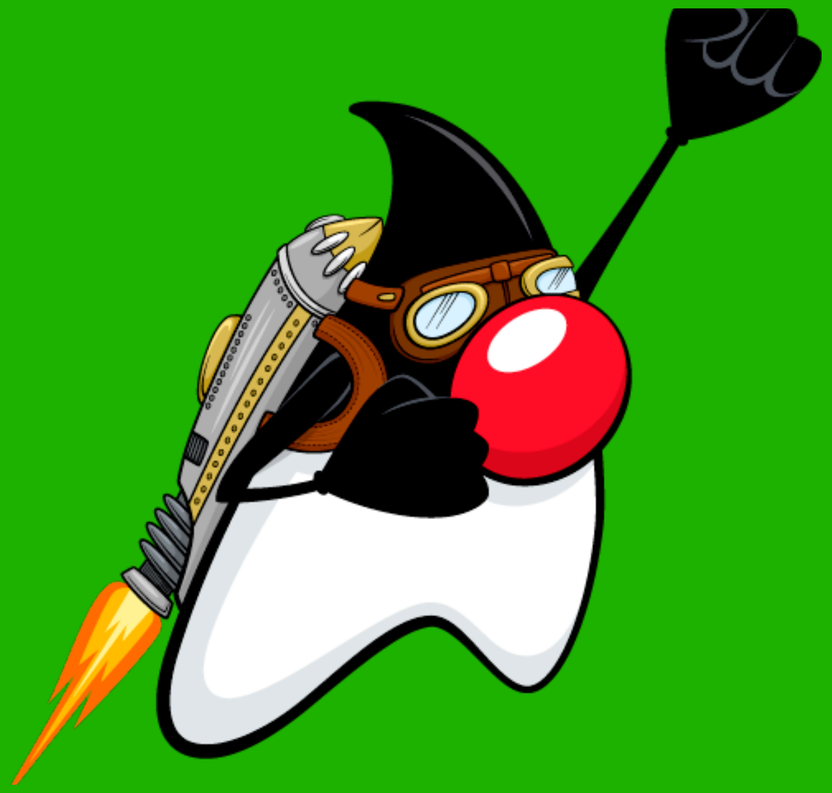
Prepare Step



Source: https://jreleaser.org

Prepare

- Generates files required by Packagers such as Homebrew.
- These files will be generated from templates existing in your project at a configured location (the templateDirectory of each packager) and default templates bundled in the JReleaser distribution.
- These template files rely on Name Templates to parameterize their contents.
- Prepared files will be placed at `${basedir}/out/jreleaser/${distributionName}/${packager}/prepared`

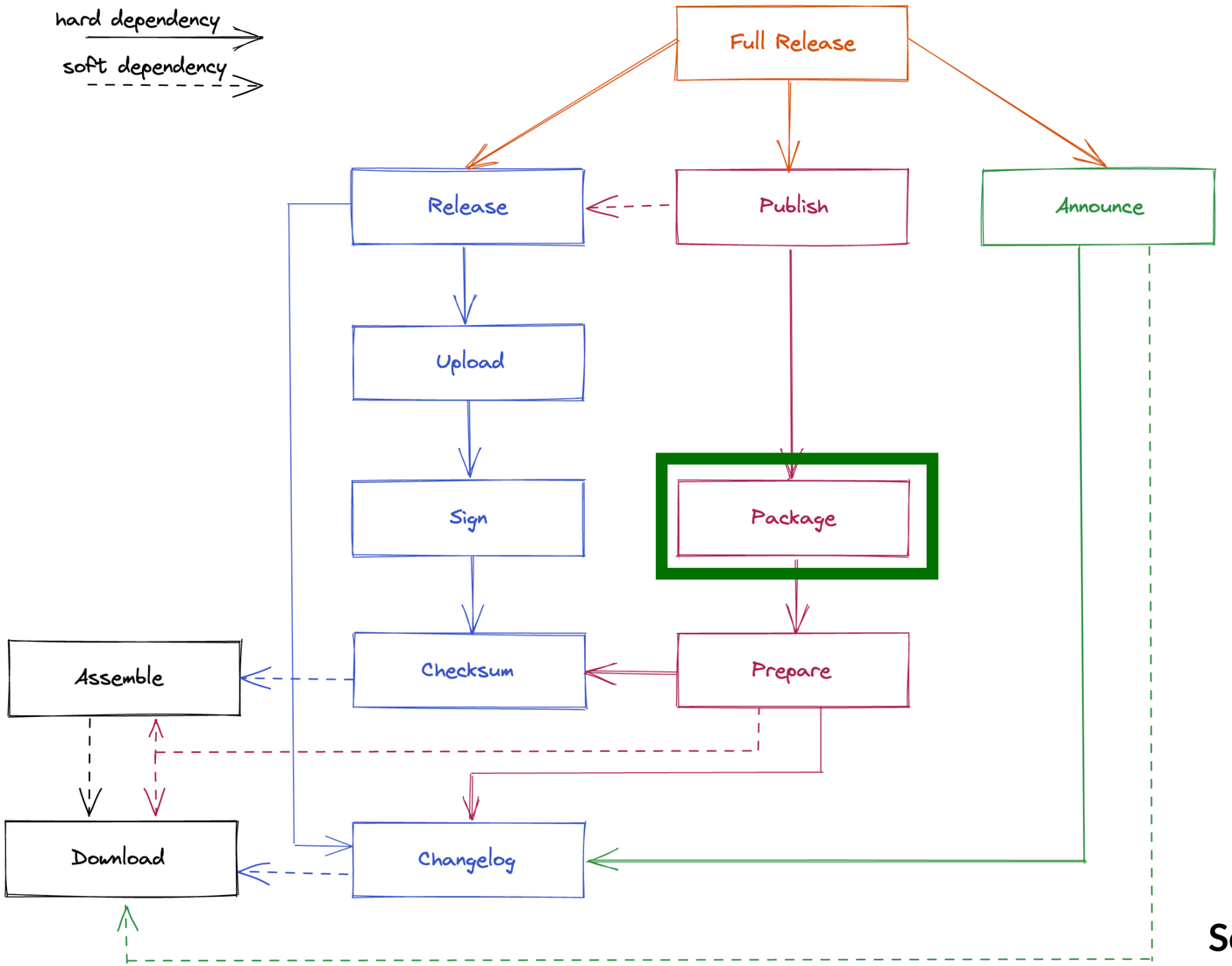


Package

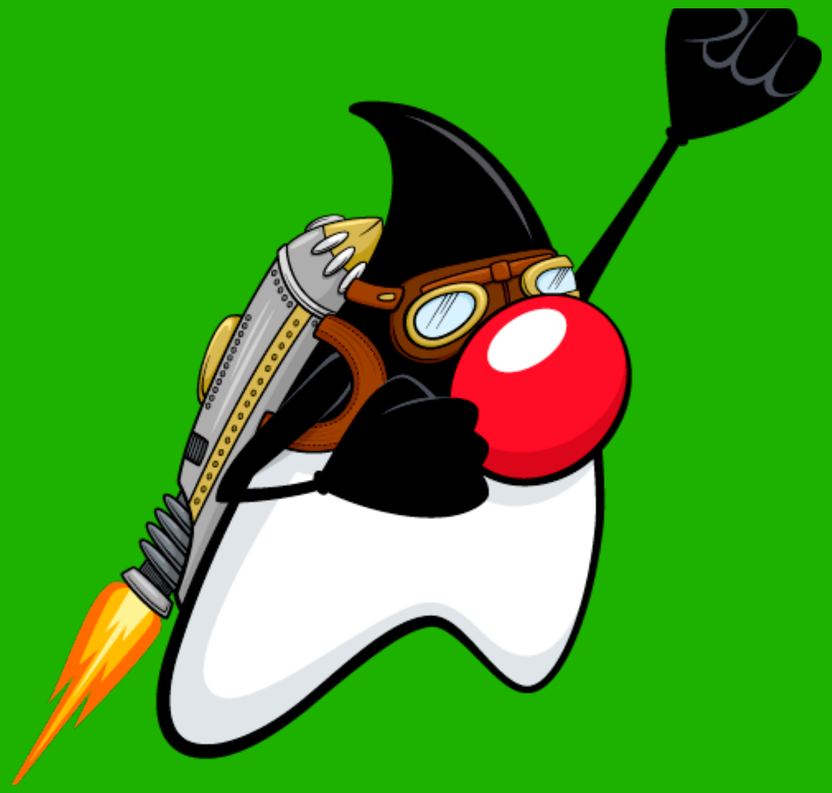
Package

- Processes the files created by the Prepare step *to create specific packages*.
- If no additional processing is required, then it will merely copy the files over from their matching prepared directory to their respective package directory
- Packaged files will be placed at `${basedir}/out/jreleaser/${distributionName}/${packager}/prepared`

Package Step



Source: https://jreleaser.org

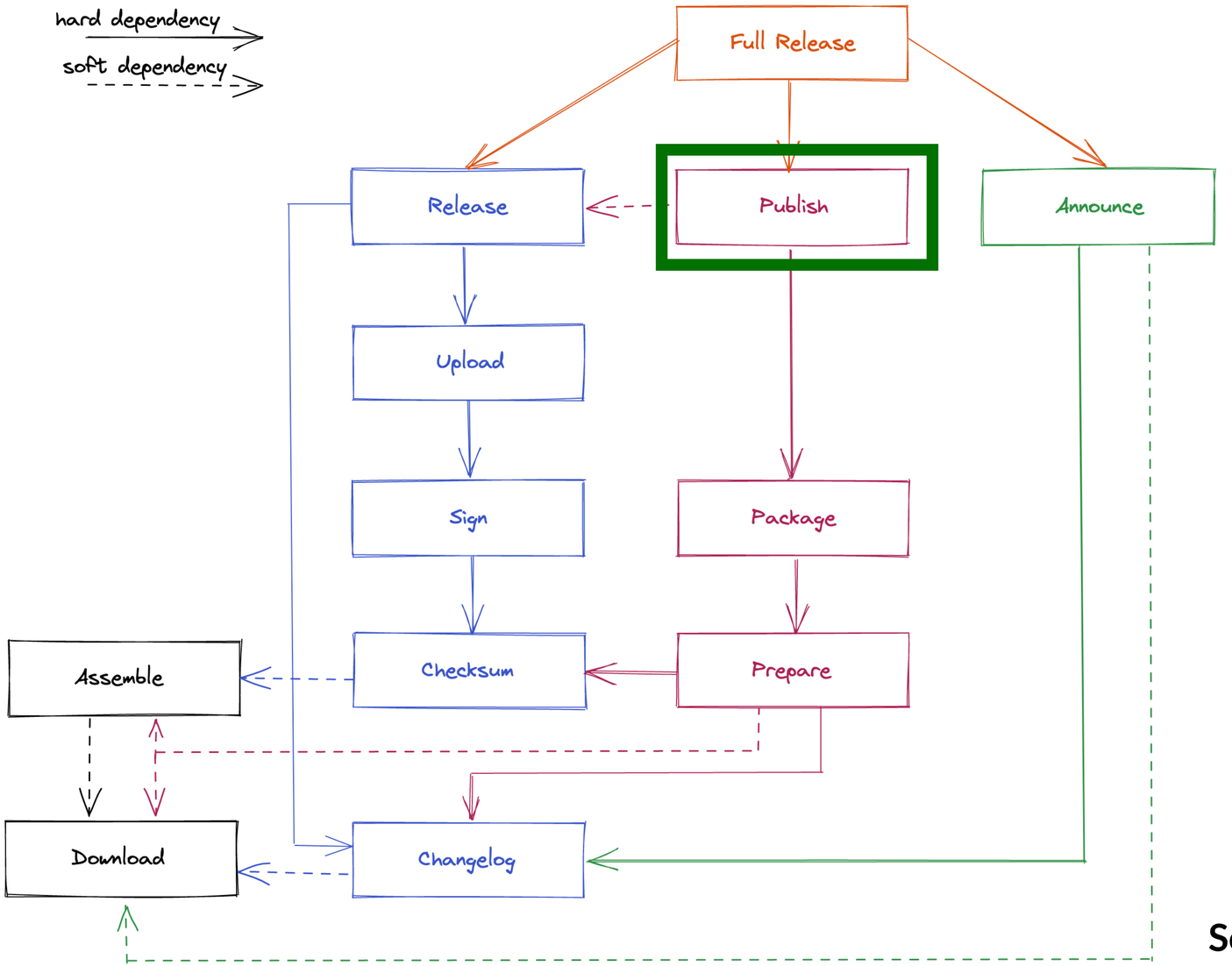


Publish

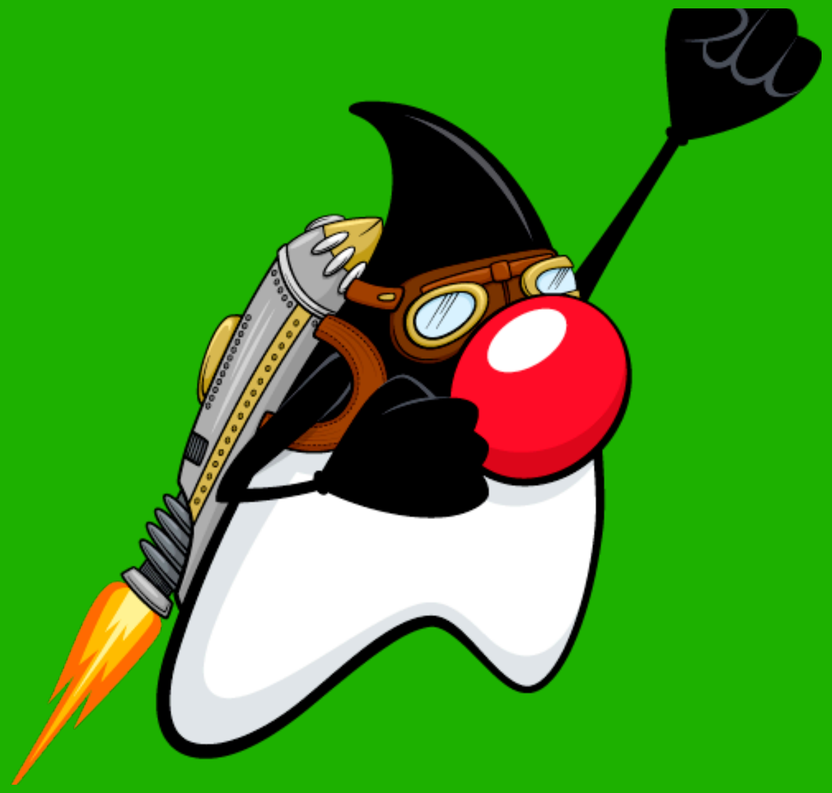
Publish

- Publishes packaged files to their respective destinations (e.g. Homebrew)
- This step depends on the *package* step
- Does not explicitly depend on Release however some packagers such as Snap require downloading distribution files to locally build a snap.

Publish Step

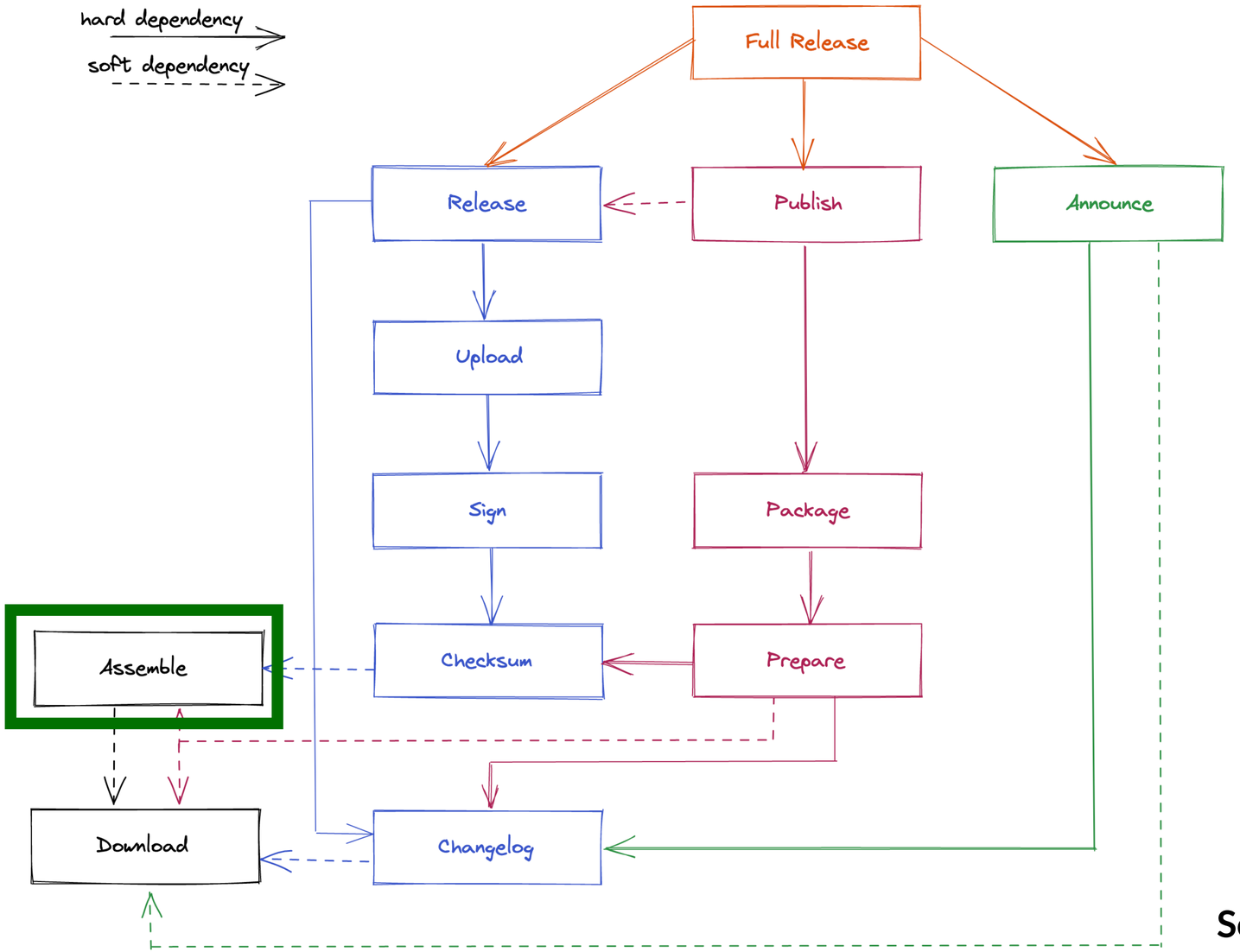


Source: https://jreleaser.org



Assemble

Assemble Step



Source: https://jreleaser.org

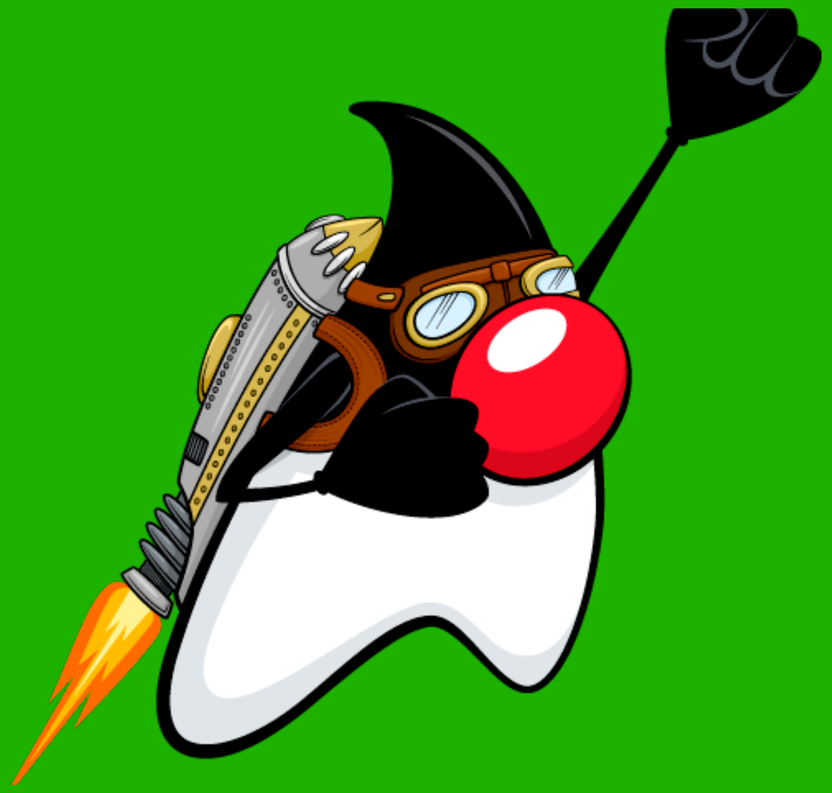
Assemble

- Assembles distributions such as Jlink and Native Image.
- Outputs will automatically configure/update matching named distributions for *Prepare*, *Package*, and *Publish*.
- Assemblers will configure a matching distribution by name if it exists and if the assembler exports its artifacts (true by default)
- Otherwise they'll create a new distribution with their respective name
- *Assemblers require to be run first and separately before running any publishers*
- Assembling can be done by your build tool of choice (Maven, Gradle) and JReleaser can make use of those distributions

Assembling

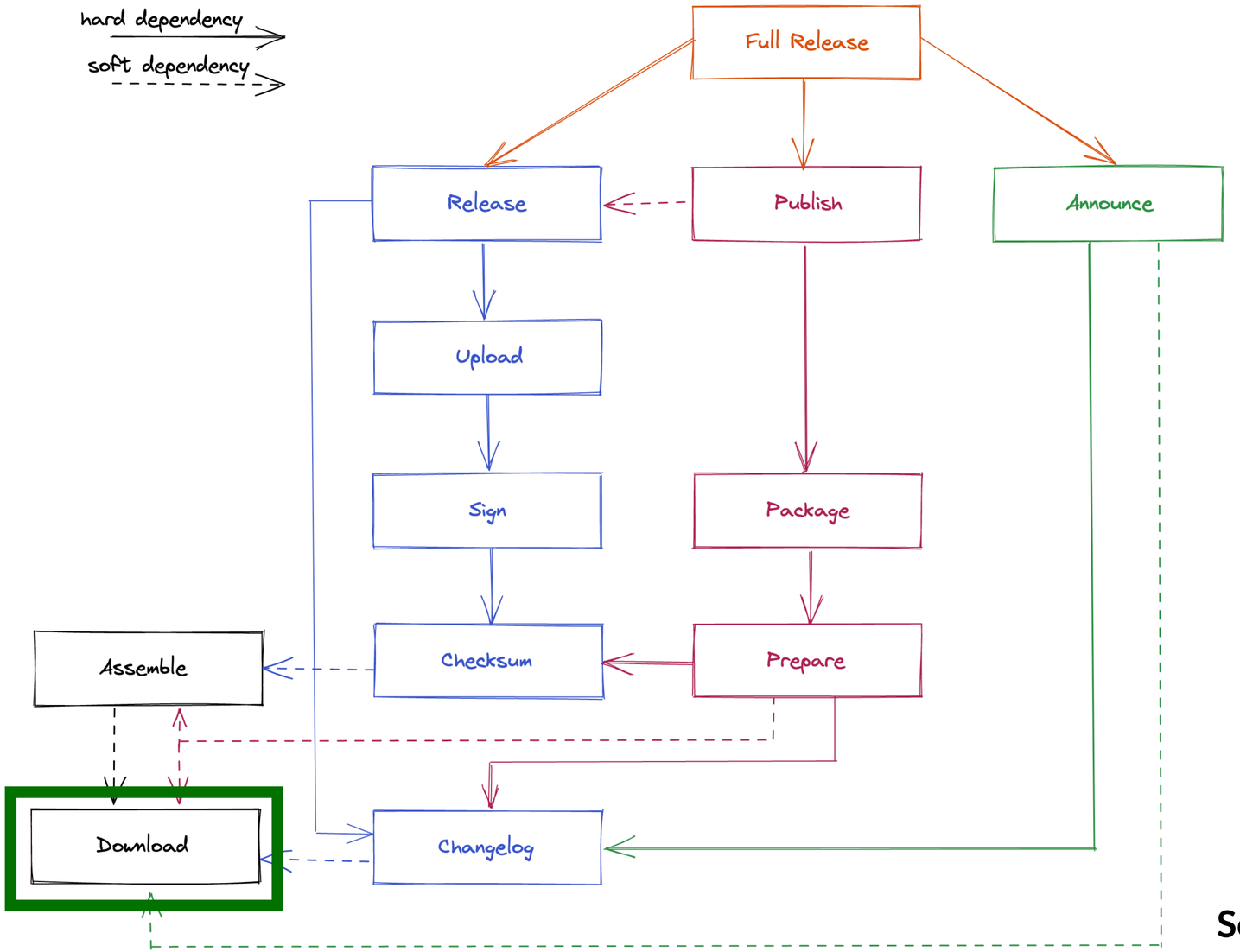
Distributions may be assembled using your build tool of choice, also with any of the following assemblers:

- **Archive** - Binary distribution archive
- **JLink** - Java tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.
- **JPackage** - Java tool tht enables you to generate installable platform specific packages for modular and non-modular Java applications. Uses JLink
- **Native Image** - Creates a Native Image binary using GraalVM Native Image.



Download

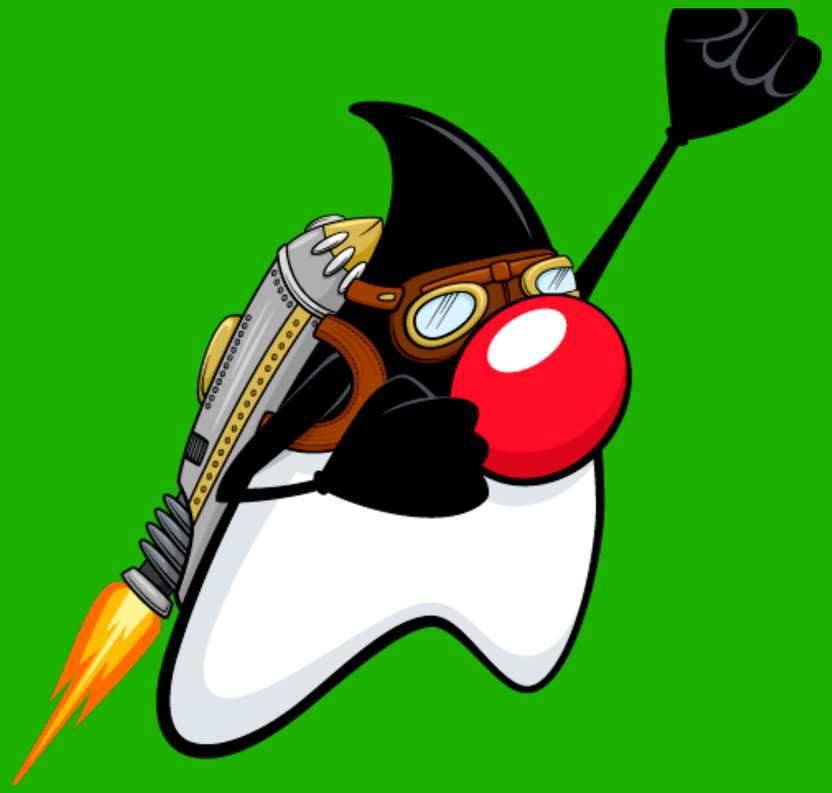
Download Step



Source: https://jreleaser.org

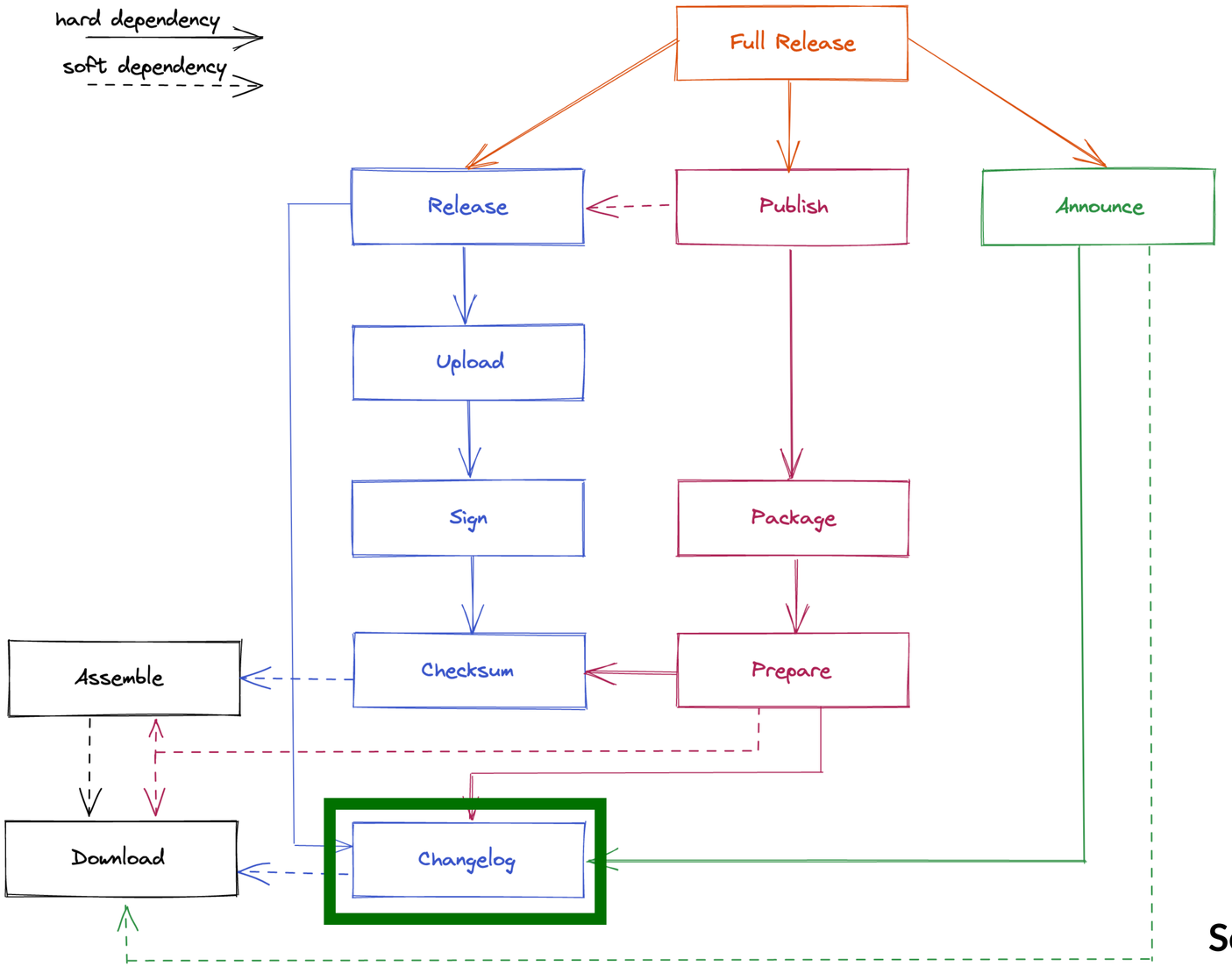
Download

- Artifacts and files may be downloaded from configured locations
  - FTP
  - HTTP
  - SCP
  - SFTP
- This step must be run separately from the others



Changelog

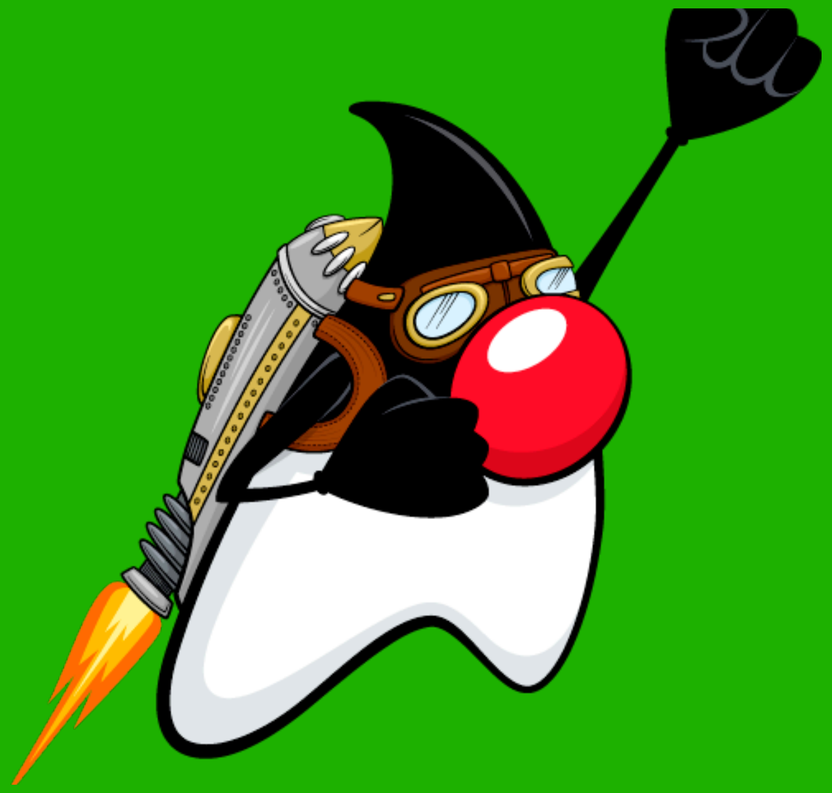
Changelog Step



Source: https://jreleaser.org

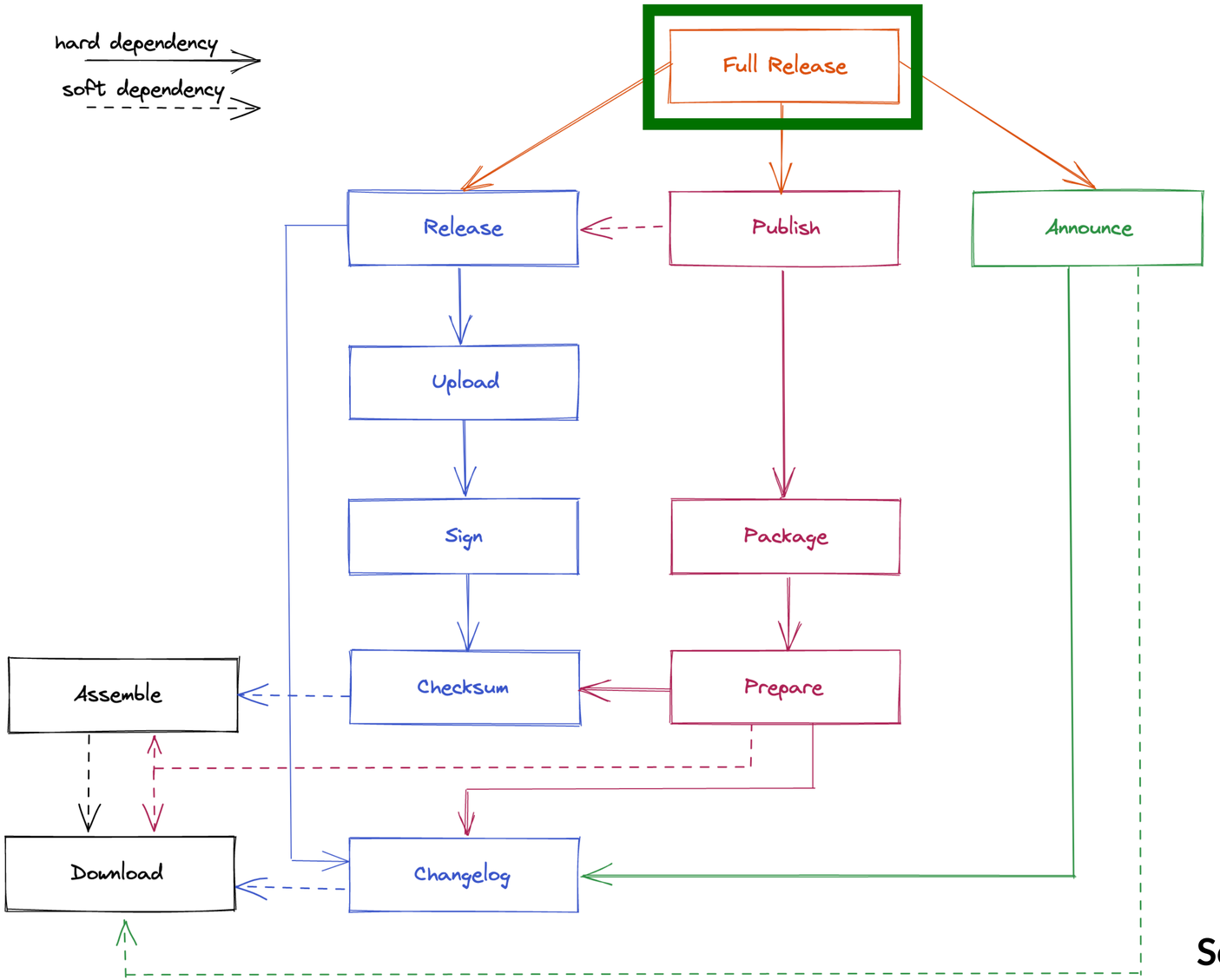
Changelog

- Creates the changelog
- Either takes
  - A supplemented external file
  - Calculates it based on the last tag that matches the configuration.



Full Release

Full Release Step

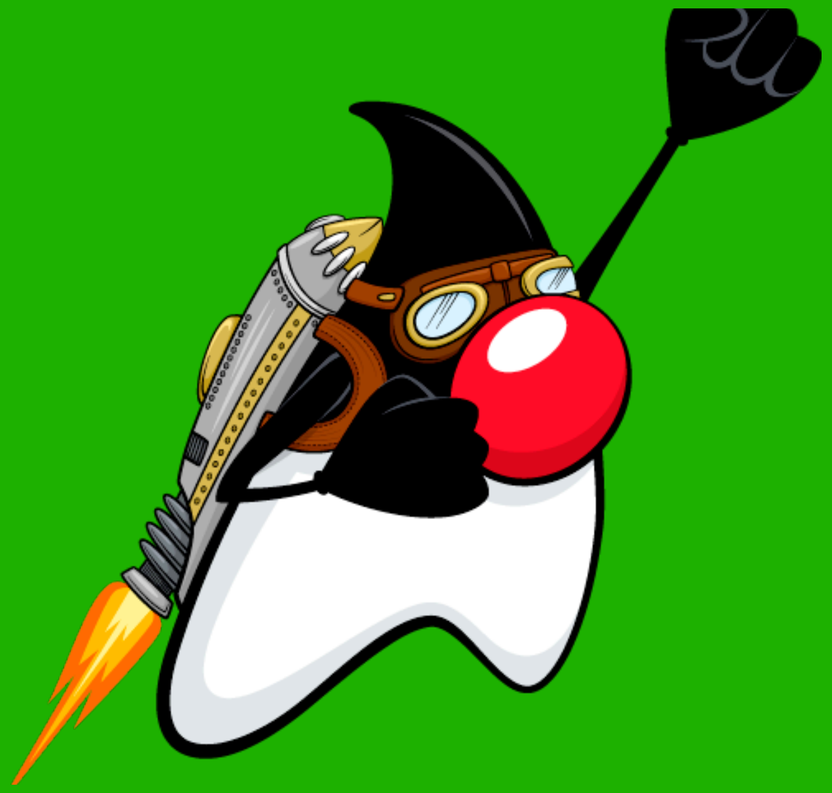


Source: https://jreleaser.org

Full Release

Executes all steps in single session in the following order:

- Changelog
- Checksum
- Sign
- Upload
- Release
- Prepare
- Package
- Publish
- Announce

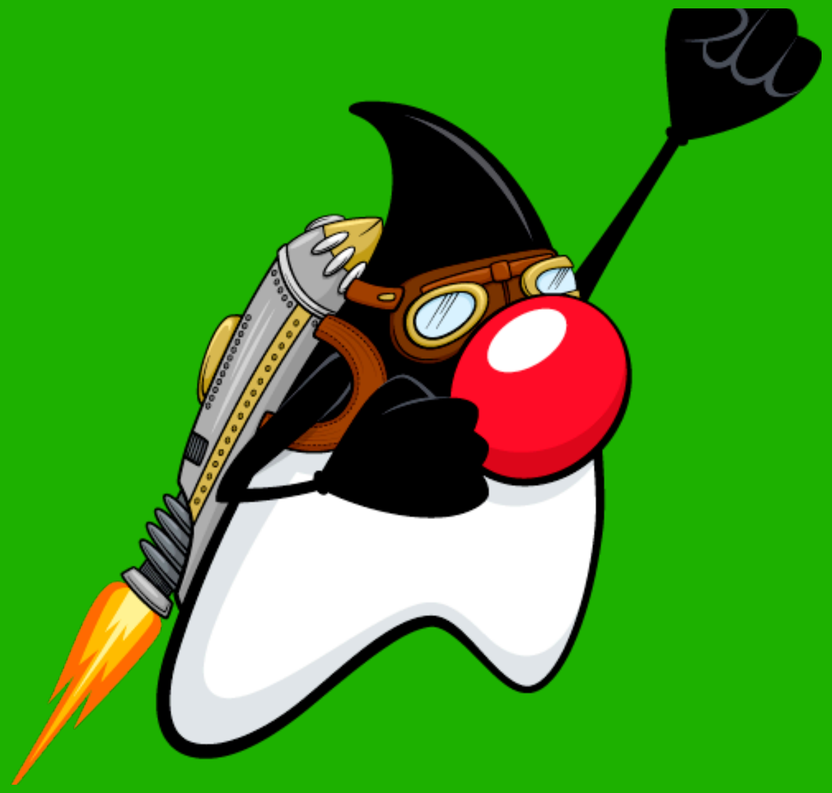


Continuous Integration

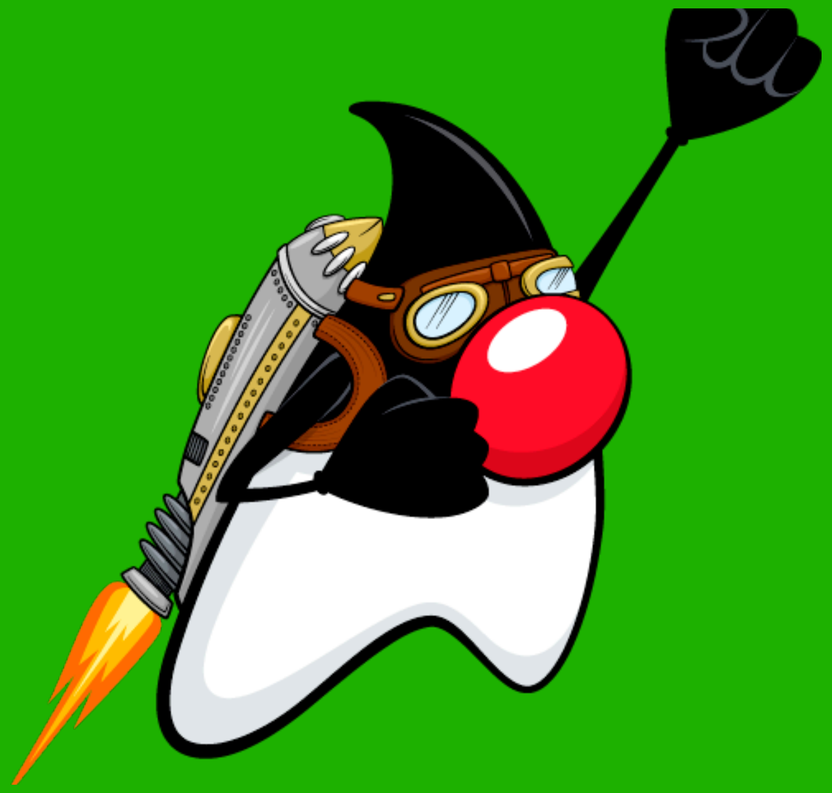
Continuous Integration

JReleaser can be run as part of a CI pipeline. The following options are currently available, with more to come in the future:

- Buddy
- Buildkite
- Circle CI
- Cirrus CI
- Codefresh
- Codeship
- Drone
- Github Actions
- GitLab CI
- Jenkins
- Semaphore
- TeamCity
- Travis CI
- Wercker



Demo: Continuous Integration



Conclusion