Degasolv Documentation

Release 2.2.0

Daniel Jay Haskin et. al., see the AUTHORS.md file.

Nov 18, 2019

Contents

Star Degasolv on Github:

Fork Degasolv on Github:

CHAPTER 1

Why Degasolv?

Degasolv is a generic dependency resolver that exists independent of programming languages or systems. You can use it to easily declare the existence of files that your build depends on, version them, and retrieve the URLs of files which are of the correct versions. You can easily use these URLs in your builds to download everything the build needs.

Since Degasolv is a dependency resolver that is relatively technology-agnostic, you can declare dependencies between components that are not of the same technology. You can declare that a DLL depends on a pip package, or that in order to use an NPM package, a certain ruby gem file must be present as well.

Often when building software, multiple different components from multiple different teams must be used to create a larger build artifact. These components are "released" by different teams, and these teams each use different technologies, or the teams themselves do not use a dependency manager. As a build engineer, it's your job to bring all of these components together to make the build work. Degasolv helps you do this. Some exmamples of people who might use degasolv are below. For a more detailed example, see A Longer Example.

- Sravan, a build engineer, is repsponsible for the deployment pipeline of his company's cloud offering. There are several components created by different teams within his company, using totally different technologies: docker images, VM templates, and even PXE files are used to deploy different parts of the cloud stack. Each come from a different team, and each are released on different schedules. Sravan uses degasolv to version and track these artifacts and define relationships between them. He uses a CI build to get the URLs of every VM template, docker image, and PXE file, and places all these in a zip file which represents his entire cloud stack as a standalone build artifact, which can be promoted through environments and run through QA with minimal effort.
- Sheila, a build engineer, is responsible for the build of an microsoft installer which installs her company's product. The installer contains python components, native code components, and ruby components. The installer also takes files from a self-extracting tarball created by a third-party vendor, which has its own dependencies. With degasolv, Sheila can track all of these files independently of where the files are actually located, and she can use degasolv to tell her the exact files she needs for her build.
- Daryl, a build engineer, is responsible for building a native code library (DLL or SO) file from the native code output from multiple teams. Each team releases their code as a zip file or tarball, but haven't realy adopted a formal dependency resolver in their builds. Further, they are building code for microsoft as well as linux, and MSBuild files only support relative paths when referring to dependencies in a build. Daryl puts the zip archives on his NAS, and uses degasolv to resolve the dependencies of the library. Degasolv returns URLs for all the

packages to Daryl, who then uses them in a script to download them and place them in the directories where they can be found by the MSBuild files. The build works flawlessly ;)

CHAPTER 2

Get Degasolv

2.1 Download & Run

Degasolv comes in the form of a `.jar` file, downloadable from GitHub.

As of version 1.8.0, it also comes in the form of an RPM or Debian package.

To get the RPM, add the CentOS bintray repository:

```
wget https://bintray.com/degasolv/centos/rpm -O bintray-degasolv-centos.repo
sudo mv bintray-degasolv-centos.repo /etc/yum.repos.d/
yum clean all
yum makecache
```

To get the debian package, add the Ubuntu bintray repository:

```
curl -L https://bintray.com/user/downloadSubjectPublicKey?username=degasolv | \
    sudo apt-key add -
echo "deb https://dl.bintray.com/degasolv/ubuntu stable main" | \
    sudo tee -a /etc/apt/sources.list.d/bintray-degasolv-ubuntu.list
```

To use it, you need java installed. Degasolv can be run like this:

```
java -jar ./degasolv-<version>-standalone.jar
```

Or, if you are using an OS package, it can be run simply like this:

```
degasolv
```

2.2 Code

Degasolv lives out on Github.

2.3 Support & Problems

If you have a hard time using Degasolv to resolve dependencies within builds, it is a bug! Please do not hesitate to let the authors know via GitHub issue :).

You can also talk to us using Gitter or the Google Group "degasolv-users".

2.4 Contributions

Please contribute to Degasolv! Pull requests are most welcome. Please have a look at the *Contributing Guide* first.

CHAPTER 3

Quickstart

This quickstart is meant to be illustrative. For ideas on how to use degasolv in real life, have a look at A Longer Example.

Given these artifacts:

- `http://example.com/repo/a-1.0.zip`
- `http://example.com/repo/b-2.0.zip`
- `http://example.com/repo/b-3.0.zip`

1. Generate dscard files to represent them in a degasolv respository, like this:

```
$ java -jar degasolv-<version>-standalone.jar generate-card \
    --id "a" \
    --version "1.0" \
    --location "https://example.com/repo/a-1.0.zip" \
    --requirement "b>2.0" \
    --card-file "$PWD/a-1.0.zip.dscard"

$ java -jar degasolv-<version>-standalone.jar generate-card \
    --id "b" \
    --version "2.0" \
    --location "https://example.com/repo/b-2.0.zip" \
    --card-file "$PWD/b-2.0.zip.dscard"

$ java -jar degasolv-<version>-standalone.jar generate-card \
    --id "b" \
    --version "3.0" \
    --location "https://example.com/repo/b-3.0.zip" \
    --card-file "$PWD/b-2.0.zip.dscard"
```

2. Generate a `dsrepo` file from the cards:

```
$ java -jar degasolv-<version>-standalone.jar \
    generate-repo-index \
```

(continues on next page)

(continued from previous page)

```
    --search-directory $PWD \
    --index-file $PWD/index.dsrepo
```

3. Then use the `dsrepo` file to resolve dependencies:

```
$ java -jar degasolv-<version>-standalone.jar \
    resolve-locations \
    --repository $PWD/index.dsrepo \
    --requirement "b"
```

   This should return something like this:

```
a==1.0 @ http://example.com/repo/a-1.0.zip
b==3.0 @ http://example.com/repo/b-3.0.zip
```

To see the help page, call degasolv or any of its subcommands with the `-h` option. If this is your first time using degasolv, it's recommended that you read A Longer Example.

CHAPTER 4

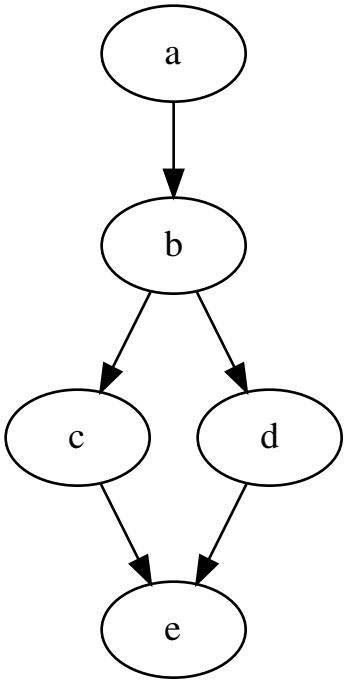
A Longer Example

4.1 Audience

This example is written for build and devops engineers who are responsible for the builds from a number of projects. These builds produce files that will here be called "artifacts". The goal in this example will be to declare dependencies between different builds and have the depenencies for each project's build downloaded automatically as part of the build. Under normal circumstances, a package manager would be used to do this; however, sometimes this is not possible. This example in particular illustrates how degasolv can be used to resolve dependencies between zip files, which do not carry dependency information.

4.2 The dependencies

In this example, suppose that you keep the artifacts for your builds, all zip files, stored on an auto-indexed HTTP server called `reposerver`, which serves the files at the URL `http://example.com/repo/`. These builds depend on the presence of artifacts from other builds to complete successfully. The dependency tree looks like this:



For example, in order to build the artifact for `a`, there must first be artifacts generated by the `b`, `c`, `d`, and `e` builds present in the build directory.

The complication here is that each project above has generated artifacts at different versions. To be short in writing, we will denote the artifact generated by the build for `a` at version `1.0.0` as `a@1.0.0`.

In our example, there was a recent breaking change to `a`. Where artifacts `a@1.9.0` worked fine with all previous versions of artifacts for `b`, the newer `a@2.1.0` only works with `b@2.3.0` or greater. Since the `2.0.0` line of `b` came out, it relies on the newer `c@3.5.0`, and the ancient-but-still-used `d`, the only version of which was published as `d@0.5.0`. The last time `d` was touched, the newest version of `e` was `1.1.0`; however, the newer `c@3.5.0` relies on the fact that the artifact for `e` must be at least at version `1.8.0` or newer. There are three published artifacts at different versions for `e`: `e@1.8.0`, `e@2.1.0`, and `e@2.4.0`. Only the `e@1.8.0` version of `e` is backwards compatible with `e@1.1.0` and so it is the only version which will satisfy all of the build-time dependencies for `a`.

4.3 Adding `e` to the degasolv repo

The first step is to build `e`, since it is at the bottom of our dependency tree. In our example, when we build `e`, we mean that we are generating the file `e-<version>.zip` using the source code for `e`. Let's say that we have as part of this build already created `e`, at the version of `1.8.0`. We might have a file called `degasolv.edn` somewhere in our source code repository for `e`. We can use this file to specify options to degasolv, including repositories, requirements, etc. of the build. The file will be simple for `e`, though, since `e` has no other dependencies. It might look like this:

```
; filename: degasolv.edn
{
```

(continues on next page)

(continued from previous page)

```
    :id "e"
    :version "1.8.0"
}
```

During the build of `e`, we push the build artifact `e-1.8.0.zip` to the `reposerver` so that it can be downloaded at `https://example.com/repo/e-1.8.0.zip`. Then, we generate a `dscard` file for `e`. This file will represent `e` in a degasolv repository. It is done like this:

```
$ java -jar degasolv-<version>-standalone.jar \
    generate-card \
    --location "https://example.com/repo/e-1.8.0.zip" \
    --output-file "e-1.8.0.zip.dscard"
```

Note that it is good practice to name the output file after the name of the file that the card will be representing in the degasolv respository.

This will create a file called `e-1.8.0.zip.dscard`. We would then copy this file up to the `reposerver`:

```
$ rsync e-1.8.0.zip.dscard user@reposerver:/var/www/repo/
```

Once the card is added to the `repo` on the repo server, a command is run on the server to generate (or update) a degasolv repository index:

```
$ ssh user@reposerver
$ cd /var/www/repo
$ java -jar ./degasolv-<version>-standalone.jar \
    generate-repo-index \
    --search-directory /var/www/repo \
    --output-file /var/www/repo/index.dsrepo
```

This command takes all of the package information from all of the degasolv card files found under `/var/www/repo` and adds this information to the repository index `/var/www/repo/index.dsrepo`. Once this is done, the package `e` is listed as available in the degasolv respository index. We can check that listing `e@1.8.0` as available in the index was successful by querying the index from any machine that can see the `index.dsrepo` file on the reposerver, like this:

```
$ java -jar ./degasolv-<version>-standalone.jar \
    query-repo \
    --repository "https://example.com/repo/index.dsrepo" \
    --query "e"
```

Supposing that multiple versions of e is in the repository, its output will look like this:

```
e==1.8.0 @ https://example.com/repo/e-1.8.0.zip
e==2.1.0 @ https://example.com/repo/e-2.1.0.zip
e==2.4.0 @ https://example.com/repo/e-2.4.0.zip
```

We can see that the version of `e` we were building, namely `1.8.0`, is now in the repository index. We now know that the repository index has been properly updated.

4.4 Adding `d` to the degasolv repo

In our example, `d` is ancient, and not built anymore in our environment; however, it is still used in other builds. We will not use a `degasolv.edn` file for it, because there is nowhere to commit such a file to source. We will simply generate a `dscard` file for it using command line options:

```
$ java -jar degasolv-<version>-standalone.jar \
    generate-card \
    --id "d" \
    --version "0.5.0" \
    --location "https://example.com/repo/d-0.5.0.zip" \
    --requirement "e>=1.00,<2.0.0" \
    --output-file "d-0.8.0.zip.dscard"
```

Note that we can either use command-line options or config file keys to specify the information that degasolv needs.

We then copy the newly created `d-0.5.0.zip.edn` file up to the server and use it to update the repository index in the same way as for `e` above.

4.5 Adding `c` to the degasolv repo

The `c` artifact (zip file) represents a project that is being actively built and developed, so we will create a `degasolv.edn` file and commit it to the source repository for `c`. The build for `c` relies on the `e` artifact being present, so we will resolve that dependency before we start the build for `c`. Then, when we build the `c` project, we will create its corresponding degasolv card file as part of the build, like we did with `e`.

First, we commit its `degasolv.edn` file to source code. It might look like this:

```
; filename: degasolv.edn
{
    :id "c"
    :version "3.5.0"
    :requirements ["e>=1.8.0"]
    :repositories ["https://example.com/repo/index.dsrepo"]
}
```

As mentioned earlier, `c` needs the `e` artifact in order to build. We will use `degasolv` as part of `c` build script to download the most recent version fitting the requirement for `e` like this:

```
$ java -jar degasolv-<version>-standalone.jar \
    resolve-locations
```

This command is run from the same directory where `degasolv.edn` resides. It will return output looking something like this:

```
e==1.8.0 @ https://example.com/repo/e-1.8.0.zip
```

We can use this output in a script to download and unzip the zip file so that it can be used as part of the build for `c` like this:

```
#!/bin/sh

java -jar degasolv-<version>-standalone.jar -c ./degasolv.edn \
    resolve-locations | while read pkg
do
  spec=$(echo "${pkg}" | awk -F ' @ ' '{print $1}')
  name=$(echo "${spec}" | awk -F '==' '{print $1}')
  version=$(echo "${spec}" | awk -F '==' '{print $2}')
  url=$(echo "${pkg}" | awk -F ' @ ' '{print $2}')
  curl -o ${name}-${version}.zip -L ${url}
  unzip ${name}.zip
done
```

This stanza can be used in a build script to download all of the dependencies for `c` and unzip them in the current directory.

At the end of the build for `c`, we can create the degasolv card file for `c` like this:

```
$ java -jar degasolv-<version>-standalone.jar \
    generate-card \
    --location "https://example.com/repo/c-3.5.0.zip" \
    --output-file "c-3.5.0.zip.dscard"
```

Then we upload this file to our http server and use it to update the `index.dsrepo` degasolv repository index file in the same way as what we did during the build for `e`.

Let us now suppose that we have repeated these steps for the build artifacts of `b`. Then all of the projects except for `a` which are mentioned at the beginning of this example will have had artifacts built from their builds and entries created in the degasolv respository index for their artifacts.

4.6 Building `a`

Now suppose that we are building `a`. In our example, the build artifact for `a` need not be uploaded to the zip file repository, because `a` represents our final product, and the build for `a` will generate an artifact that will be handed off to Project Management or Ops for later release. We don't need it for any other builds. While we are not (in this trivial example) not interested in uploading it to the repo, we are interested in resolving its dependencies, downloading them, and using them to build the final product.

Just like some of our previously described builds in this example, we will put a file called `degasolv.edn` in the root of the git repository associated with building `a`. It might look like this:

```
; filename: degasolv.edn
{
    :id "a"
    :version "2.1.0"
    :file-name "a-2.1.0.zip"
    :requirements ["b>2.0"]
    :repositories ["https://example.com/repo/index.dsrepo"]
}
```

Then, as in the script used to build the artifact for `a`, we resolve its dependencies and download them, just as we did when we built `e`:

```
#!/bin/sh

java -jar degasolv-<version>-standalone.jar -c ./degasolv.edn \
    resolve-locations | while read pkg
do
  spec=$(echo "${pkg}" | awk -F ' @ ' '{print $1}')
  name=$(echo "${spec}" | awk -F '==' '{print $1}')
  version=$(echo "${spec}" | awk -F '==' '{print $2}')
  url=$(echo "${pkg}" | awk -F ' @ ' '{print $2}')
  curl -o ${name}-${version}.zip -L ${url}
  unzip ${name}.zip
done
```

This will resolve all of the dependencies for `a`, download them, and unzip them. The rest of the build process for `a` can then continue as normal.

CHAPTER 5

Changelog

All notable changes to this project will be documented here.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

5.1 Unreleased

5.1.1 Added

5.1.2 Changed

5.1.3 Fixed

5.2 2.2.0

5.2.1 Added

- Configuration file tower: Degasolv looks for config files in pre-defined locations on the file system if they exist, see *Default Configuration Files*.
- Environment variable support: environment variables are consulted first, but are merged into the option map AFTER config file material and BEFORE any given command line options, see *Environment Variables* and *How Options are Gathered*
- Added support for HTTP/HTTPS authentication using basic, oauth2 token and header methods, see *A Note on Specifying Files*.

5.2.2 Changed

- Removed deprecated, unused function `degasolv.util.assoc-conj`

5.2.3 Fixed

- Option packs are expanded at the level upon which they are defined, then the options are merged together. This seemed like a bug that needed fixing. See *Option Packs* for more information.

5.3 2.1.0

5.3.1 Added

- Added "version suggestion", a performance enhancement allowing for minimum version selection
- Added the `:index-sort-order` option to `generate-repo-index`, allowing users to specify `ascending` or `descending`. Previously only `descending` was supported. With this new option, users will be able to use Degasolv in a minimum version selection configuration.
- If the reason for failure in the `resolve-dependencies` function is `:present-package-conflict`, add a key `:package-present-by` with value as either `:found` meaning the package was in conflict with a package found during resolution, or with value as `:given` meaning the package was in conflict with a package given via the parameter `present-packages`.

5.3.2 Changed

- Removed the deprecated functions `->requirement`, `->package`, and `->version-predicate` from usage in the code base.
- Removed `dbg2` macro in favor of keeping `dbg`

5.3.3 Fixed

- Standardized capitalization of the word "Degasolv" to be title case for consistency in the documentation unless it is in a code snippet.
- Fix #16
- Fix spec for package id's. This should have the effect of enforcing that names should not have `>`, `<`, `!`, `=`, `,`, `;`, or `|` characters in them. This simply has the effect of changing the error message, as this was never allowed but handled poorly.

5.4 2.0.0

5.4.1 Added

- Documentation saying what return codes are given and what they mean.
- For #15, added ability to specify output format for `display-config`.
- Added *3rd Party Licenses* document

5.4.2 Changed

- In the docs, `java -jar degasolv-<version>-standalone.jar` changed to `degasolv` with added note for clarity
- For #13, return code for `resolve-locations` changed to 3 when dependency resolutions occur to distinguish them from normal "you got the argument string wrong" errors
- Default `--{enable|disable}-error-format` set to `enabled` for *resolve-locations* and *query-repo*.
- Default for `--list-strat` option for *resolve-locations* set to `lazy`, a much saner default.
- Option pack `v1` *added* to help administrators keep compatibility with version 1 of Degasolv if required.
- Default for the `--version-comparison` option when `--package-system` is `degasolv` set to `semver` for *generate-repo-index* (option *here*), *resolve-locations* (option *here*) and *query-repo* (option *here*).
- Removed less-than-useful warning about absent config files.

5.4.3 Fixed

- Fixed #14, "Degasolv pulls in X"
- Fixed bug where `display-config` didn't allow the user to specify valid options for other things, now it does
- Fixed bug where index.dsrepo didn't generate anything except an empty map inside the file. This was *completely* broken.
- Fixed #6, "If one config file fails to load, the rest do as well"
- Fixed #9, "Heading for 'Specifying Subproc Executable' is wrong in docs"
- Fixed #10, "How do you specify requirements of a package (deps) in the output of a subproc to Degasolv?"

5.5 1.12.1

1.12.1 was strictly a docs release. No code changes were made and no runnable artifacts were created.

5.5.1 Added

- Added authors file to docs

5.5.2 Fixed

- Fixed changelog so that the 1.12.0 release was present in the document
- Fixed package system subheaders in command reference

5.6 1.12.0

5.6.1 Added

- Added the `--{enable|disable}-error-format` options to *resolve-locations* and *query-repo*.

- Added the `--package-system subproc` option, together with its *Specify Subproc Package System Executable* and *Specify Subproc Package System Output Format* options.
- Added the `--json-config` *global option* allowing users to use JSON config files
- Added the `--list-strat` option to *resolve-locations*, allowing users to have their dependencies listed in a sane order.

5.6.2 Changed

5.6.3 Fixed

- JSON/EDN output for `query-repo` erroneously listed the subcommand as `resolve-locations`. Fixed.

5.7 1.11.0

5.7.1 Added

- Added the `--meta` *option* to *generate-card*
- Added metadata a la `--meta` to the `apt` *package system* (experimental)
- Added the `edn` *output format option* for the `resolve-locations` *subcommand*
- Added the `--output-format` *option* to the *query-repo* command

5.7.2 Changed

- Overhauled the documentation option look and feel; ensured that options themselves had a linking target (header)

5.8 1.10.0

5.8.1 Added

- Added the `--output-format` *option* to *resolve-locations*

5.9 1.9.0

5.9.1 Added

- Added the *pessimistic greater-than* comparison operator `><`.

5.9.2 Fixed

- Removed validation from the config file option, allowing it to be a URL or anything else.
- If no arguments are given, the help screen is now printed instead of a wierd error.

5.10 1.8.0

5.10.1 Added

- Distribution is now done via RPM and Debian package as well as JAR fil
- Added the `--version-comparison` option to *generate-repo-index* (option *here*), *resolve-locations* (option *here*) and *query-repo* (option *here*), allowing the user to specify which version comparison algorithm is used.
- Added the `--search-strat` option to *resolve-locations*, allowing users to select breadth first search or depth first search during resolution
- Added the *matches* operator (`<>REGEX`) which matches a version against a regex
- Added the *in-range* operator (`=>V`) which matches a version against a certain range of indexes
- Added the ability to specify `--present-package` multiple times using the same package name, but different versions. This is useful for when the `:conflict-strat` is set to `inclusive`.
- Added tests testing to make sure that unsuccessful runs generate the proper error messages.

5.10.2 Changed

- Reorganized the unit tests.
- Alphabetized the options for `generate-card`.
- Alphabetized the options for `generate-repo-index`.

5.10.3 Fixed

- Fixed bug wherein if the conflict strategy is set to `:inclusive` and a package satisfying a requirement is already found or present, it is used instead of finding a new one.
- Fixed CLI of *display-config* so that it actually works as advertised, LOLZ
- Fixed the CLI output of `--help` so that default values of options are shown again :)
- Refreshed the CLI output of `--help` for all the subcommands as posted in the docs

5.11 1.7.0

5.11.1 Added

- Added `--option-pack`, the ability to *specify multiple options at once*

5.11.2 Fixed

- Fixed how default options work, they no longer override stuff found in the config file (ouch)
- Fixed output of printed warning when configuration file is not used

5.12 1.6.0

5.12.1 Added

- Formatted docs better on the front page for PDF purposes
- Add ability to use any (long) option on the command line in *display-config*

5.12.2 Improved

- Memoized core Degasolv package system repository function (should speed the resolver up a bit)
- Changed apt reop function from filtering a list to lookup in a map, increasing its speed

5.13 1.5.1

5.13.1 Added

- In just ~15 seconds, it slurps in a rather large apt repository Packages.gz file. In another ~45 seconds, it resolves the ubuntu-desktop package, spitting out a grand total of 797 packages with their locations.

5.13.2 Fixed

- While using the apt data and package system to profile Degasolv, I found some rather nasty bugs. This release fixes them. This tool is now ready for prime time.

5.14 1.5.0

5.14.1 Added

- Added the `--disable-alternatives` *option* and the `--enable-alternatives` *option* for debugging purposes.

5.15 1.4.0

5.15.1 Added

- Added the `--present-package` *option* and the `--package-system` *option* to the *resolve-locations* subcommand. This was so that Degasolv could be profiled using apt package repos (real-world data) and thereby have its performance optimized.

5.16 1.3.0

5.16.1 Added

- Add standard input as a file type. All options which take a file name may now have - given as the filename, to specify that standard in should be used.

5.17 1.2.0

5.17.1 Added

- Added the ability to specify multiple configuration files, thus allowing for site-wide configuration.

5.18 1.1.0

5.18.1 Added

- Added the `--conflict-strat` *option* to the *resolve-locations* subcommand.
- Added docs and tests.

5.19 1.0.2

- This isn't the first release, but for the purposes of these docs, it is :D

CHAPTER 6

Degasolv Command Reference

This guide describes the Degasolv CLI, what subcommands and options there are, and what they are for. It also describes how to specify options.

6.1 Some Notes on Versions

- On a best-effort basis, features have had the version that they first appeared associated with them in this guide.
- Anything tagged with version 1.0.2 *really* means "1.0.2 or earlier". The history gets shaky before that :)
- The first version of Degasolv (for the purposes of this guide) released was 1.0.2 .
- As of version 1.3.0, All options which take a file name may now have − given as the filename, to specify that standard in should be used.
- The earliest usable released version of Degasolv that can be recommended for use is 1.5.1 . Anything before that wasn't profiled, and had some pretty bad bugs in it.

6.2 Top-Level CLI

6.2.1 Top-Level Usage Page

Running `degasolv -h` will yield a page that looks something like this:

```
Usage: degasolv <options> <command> <<command>-options>

Options are shown below, with their default values and
  descriptions. Options marked with `**` may be
  used more than once.

-c, --config-file FILE  ./degasolv.edn  Config file location **
-j, --json-config FILE                  JSON config file location **
```

(continues on next page)

(continued from previous page)

```
-k, --option-pack PACK                   Specify option pack **
-h, --help                               Print this help page

Commands are:

  - display-config
  - generate-card
  - generate-repo-index
  - resolve-locations
  - query-repo

Simply run `degasolv <command> -h` for help information.
```

Note: In this guide, for brevity, the reference is presented as if the command to execute Degasolv were simply `degasolv` rather than the more correct `java -jar degasolv-<version>-standalone.jar`. A bash or batch script can easily be made to turn one command into the other, and the change was made to the former form for clarity.

6.2.2 A Note on Specifying Files

As of version 1.3.0, The whenever an option takes a file in Degasolv, the user can actually specify one of three things:

1. An `http://` or `https://` URL. Prior to version 2.2.0, no authentication was supported. As of version 2.2.0, authentication can be specified in one of three ways:
    1. **HTTP Basic Authentication**: You can specify a URL-encoded username and password to use HTTP basic authentication by separating the username and password via a `:` (colon) character and put the entire thing before the `@` character in the URL. For example:

        ```
        https://username:password@example.com/...
        ```

    2. **OAuth2 Token Authentication**: You can specify a URL-encoded OAuth2 token by simply specifying one string before the `@` in any given URL, without any separators, like this:

        ```
        https://thisisthetoken@example.com/...
        ```

    3. **Header-Based Authentication**: You can specify a custom HTTP header, together with its URL encoded value, by separating the header name from the value of the header with an = (equals) sign before the `@` in any given URL, like this:

        ```
        https://X-Auth-Token=feefiefofum@example.com/...
        ```

        This would yield a HTTP GET request with the following header:

        ```
        X-Auth-Token: feefiefofum
        ```

    As of version 2.2.0, query strings as part of the HTTP URL are also supported.
2. A `file://` URL.
3. A filesystem reference.
4. The character `-`, signifying standard input to the Degasolv process.

This is true for options of Degasolv and options for any of its subcommands.

6.2.3 Explanation of Options

There are lots of options to degasolv and a few ways in which to specify them. This section details the ways by which they should be specified.

Global Options

Degasolv parses global options before it parses subcommands or the options for subcommands; therefore, global options need to be specified first. If any option, whether global or for a subcommand is given incorrectly, the program exits with a return code of 1.

Environment Variables

Every option in Degasolv has a corresponding environment variable which, if set, will be consulted for the value of that option. Each option in this document will have its corresponding environment variable listed next to it.

- Options which take a boolean value must be specified as `true` or ``false`, as in ``export DEGASOLV_ALTERNATIVES=true`.
- Options which take a list will be specified as a single string of values separated by the caret (^) character, as in `export DEGASOLV_REQUIREMENTS=a^b^c`.
- The `:meta` option is the only option that takes a map or dictionary of values. In this option, keys and values are separated by the equals sign (=) and the list of key/value pairs are also separated by the caret character, as in `k=v^k=v^k=v...`

Note: The environment variables and their formatting will be listed for the options of all the subcommands in this document; however, **environment variables can only be used with Degasolv version 2.2.0 or greater.** This point bears special emphasis. Lots of config options say they were released in earlier versions. This is true; however, the only format of config file available for use was the EDN config file type before version 1.12.0 of Degasolv.

Using Configuration Files

Configuration files may be specified at the command line before specifying any subcommands, or in the `DEGASOLV_CONFIG_FILES` and/or the `DEGASOLV_JSON_CONFIG_FILES` environemnt. The config file structure is designed so that any command-line option may be set in the config file instead, and vice versa. More information can be found at *edn-config* and *json-config* below.

In addition, config files may be specified either in the EDN format or JSON format. Multiple config files may be specified. "Mixing and matching" of JSON and EDN config files is supported. For more information, see the *Multiple Configuration Files* section.

How Options are Gathered

First, the `DEGASOLV_CONFIG_FILES` and `DEGASOLV_JSON_CONFIG_FILES` environment variables are consulted to find any configuration files.

Next, the options in the configuration files are consulted and are merged onto each other in the order given in those variables, first EDN files and then JSON files. The last config file encountered "wins" for any given key for which multiple files specify a value.

These options are then added to and overridden by any values in environment variables, and finally added to and overridden by any values found by consulting the command line options.

Basic EDN Configuration Usage

Short option	`-c FILE`
Long option	`--config-file FILE`
Environment variable	`DEGASOLV_CONFIG_FILES=f1^f2^f3`
Version introduced	1.0.2

A config file may be specified at the command line. The config file is in the EDN format. As a rule, any option for any sub-command may be given a value from this config file, using the keyword form of the argument. For example, instead of running this command:

```
degasolv \
   generate-repo-index --search-directory /some/directory \
   [...]
```

A configuration file that looks like this could be used instead:

```
;; filename: config.edn
{
    :search-directory "/some/directory"
}
```

With this command:

```
degasolv \
  --config-file "$PWD/config.edn" \
  generate-repo-index [...]
```

Notable exceptions to this rule include options which may be specified multiple times. These options are named using singular nouns (e.g. `--repository REPO`), but their corresponding configuration file keys are specified using plural nouns (e.g., `:repositories ["REPO1", ... ]`).

So, instead of using this command:

```
degasolv \
  resolve-locations \
  --disable-alternatives \
  --present-package "x==0.1" \
  --present-package "y==0.2" \
  --repository "https://example.com/repo1/" \
  --repository "https://example.com/repo2/" \
  --requirement "a" \
  --requirement "b"
  [...]
```

This configuration file might be used:

```
; filename: config.edn
{
    :alternatives false
    :respositories ["https://example.com/repo1/"
                    "https://example.com/repo2/"]
    :requirements ["a"
                   "b"]
    :present-packages ["x==0.1"
                       "y==0.2"]
}
```

With this command:

```
degasolv \
  --config-file "$PWD/config.edn" \
  resolve-locations \
  [...]
```

Basic JSON Configuration Usage

Short option	`-j FILE`
Long option	`--json-config FILE`
Environment variable	`DEGASOLV_JSON_CONFIG_FILES=f1^f2`
Version introduced	1.12.0

Any config file option that can be specified using EDN may also be specified using the JSON format. The only difference is that a plain string should be used as the key for the config option instead of an EDN keyword.

For example, instead of using this config file:

```
; filename: config.edn
{
  :alternatives false
  :respositories ["https://example.com/repo1/"
                  "https://example.com/repo2/"]
  :id "x"
  :version "1.0.0"
  :requirements ["a"
                 "b"]
  :present-packages ["x==0.1"
                     "y==0.2"]
}
```

With this command:

```
degasolv \
  --config-file "$PWD/config.edn" \
  resolve-locations \
  [...]
```

This JSON config file may be used instead:

```
{
  "alternatives": false,
  "repositories": ["https://example.com/repo1/"
                   "https://example.com/repo2/"],
  "id": "x",
  "version": "1.0.0",
  "requirements": ["a"
                   "b"],
  "present-packages": ["x==0.1"
                       "y==0.2"]
}
```

The command to use the above JSON config file would look like this:

```
degasolv \
  --json-config "$PWD/config.json" \
  resolve-locations \
  [...]
```

Using Multiple Configuration Files

As of version 1.2.0, the `--config-file` option may be specified multiple times. As of version 1.12.0, the `--json-config` option may also be specified, and it too may be multiple times. As of version 2.2.0, configuration files can be specified using the `DEGASOLV_CONFIG_FILES` and `DEGASOLV_JSON_CONFIG_FILES` environment variables.

Degasolv processes JSON config files together with EDN config files. Each configuration file specified will get its configuration merged into the previously specified configuration files, whether those files be EDN or JSON. The exception is for environment variables; the EDN files specified in the environment will be consulted first, followed by the JSON config files specified in the environment, followed by any configuration files on the command line whether JSON or EDN. If both configuration files contain the same option, the option specified in the latter specified configuration file will be used.

As an example, consider the following *display-config command*:

```
DEGASOLV_JSON_CONFIG_FILES="$PWD/y.json" \
DEGASOLV_CONFIG_FILES="$PWD/x.edn" \
degasolv \
  --config-file "$PWD/a.edn" \
  --json-config "$PWD/j.json" \
  --config-file "$PWD/b.edn" \
  display-config
```

If this is the contents of the file `x.edn`:

```
{
    :conflict-strat "inclusive"
    :error-format false
}
```

And this were the contents of the file `y.json`:

```
{
    "conflict-strat": "prioritized",
    "error-format": true
}
```

And this is the contents of the file `a.edn`:

```
{
    :index-strat "prioritized"
    :repositories ["https://example.com/repo1/"]
    :id "a"
    :version "1.0.0"
}
```

And this were the contents of `j.json`:

```
{
    "id": "j",
```

(continues on next page)

(continued from previous page)

```
    "alternatives": false,
    "requirements": ["x", "y"]
}
```

And this were the contents of `b.edn`:

```
{
    :repositories ["https://example.com/repo2/"]
    :id "b"
    :version "2.0.0"
    :requirements []
}
```

Then the output of the above command would look like this:

```
{
    :alternatives false
    :error-format true
    :index-strat "priority"
    :repositories ["https://example.com/repo2/"]
    :id "b"
    :version "2.0.0"
    :conflict-strat "prioritized"
    :requirements []
    :arguments ["display-config"]
}
```

Note: The JSON config file keys and their formatting will be listed for the options of all the subcommands in this document; however, **JSON config files can only be used with Degasolv version 1.12.0 or greater.** This point bears special emphasis. Lots of config options say they were released in earlier versions. This is true; however, the only format of config file available for use was the EDN config file type before version 1.12.0 of Degasolv.

Default Configuration Files

All previous versions prior to 2.2.0 of degasolv will look for a file called `./degasolv.edn` if no other config file was specified.

As of version 2.2.0, If no configuration files are specified, they will be looked for in the following locations, if they exist, as if they were specified in the following order on the command line:

1. `${AppData}/degasolv/config.edn`
2. `${AppData}/degasolv/config.json`
3. `${HOME}/.degasolv.edn`
4. `${HOME}/.degasolv.json`
5. `./degasolv.edn`
6. `./degasolv.json`

Using Site-Wide Configuration Files

The merging of config files, together with the interesting fact that config files may be specified via HTTP/HTTPS URLs, allows the user to specify a *site config file*.

Multiple sub-commands have options which fundamentally change how Degasolv works. These are `--conflict-strat`, `--index-strat`, `--resolve-strat` and `--search-strat`. It is therefore recommended that these specific options are specified site-wide, if they are specified at all. Specifying these in a site config file, then serving that config file internally via HTTP(S) would allow all instances of Degasolv to point to a site-wide file, together with a build-specific config file, as in this example:

```
degasolv \
    --config-file "https://nas.example.com/degasolv/site.edn" \
    --config-file "./degasolv.edn" \
    generate-card
```

Also remember that config files can be specified as environment variables. For example, the above example would look like this, if environment variables were used:

```
export DEGASOLV_CONFIG_FILES="https://nas.example.com/degasolv/site.edn^./degasolv.edn
↪"
degasolv \
    generate-card
```

Here is a version of that example that uses JSON instead:

```
export DEGASOLV_JSON_CONFIG_FILES="https://nas.example.com/degasolv/site.json^./
↪degasolv.json"
degasolv \
    generate-card
```

Security Considerations

Some configuration items in Degasolv, such as URLs that point to config files and repository indexes, may have passwords or API tokens in-line in the URL. As an admin, you're going to have to figure out how you want to get sensitive information of this kind into the configuration for Degasolv to consume.

There are three main ways to do this for the purposes of security:

1. **Environment Variable: Any option in Degasolv can be specified using** environment variables. See the *environment-variables* section.
2. **Standard Input: Degasolv configuration can be specified using** standard input. This is probably the most secure and least convenient way of providing sensitive information to Degasolv. Here is an example:

   ```
   degasolv -j - << DEGASOLV_CONFIG
   <CONFIG ITEMS HERE>
   DEGASOLV_CONFIG
   ```

3. **Configuration File: This method is somewhat secure as long** as the filesystem is deemed trustworthy and as long as the proper file permissions are in place so that the credentials can only be viewed by approved users.
4. **CLI: This is not normally secure, but Degasolv leaves the decision** of what is sufficiently secure to the user, and allows sensitive information to be specified on the command line in the normal way in an effort to make extra, extra sure the tool is usable even in a firestorm (read: even in the presence of bizarre use cases).

Option Packs

Short option	`-k PACK`
Long option	`--option-pack PACK`
EDN config file key	`:option-packs ["PACK1",...]`
JSON config file key	`"option-packs": ["PACK1",...],`
Environment variable	`DEGASOLV_OPTION_PACKS="P1^P2^..."`
Version introduced	1.7.0

Specify one or more option packs. The commandline version of this option may be specified multiple times.

Degasolv ships with several "option packs", each of which imply several Degasolv options at once. When an option pack is specified, Degasolv looks up which option pack is used and what options are implied by using it. More than one option pack may be specified.

Prior to version 2.2.0, If option packs were specified both on the command line and in the config file, the option packs on the command line are used and the ones in the config file were ignored.

As of version 2.2.0, Option packs are "expanded" into the options they imply on the level in which they are specified, where in a particular configuration file, in the environment, or on the commandline. Then options are merged according to the usual rules – first configuration files are merged (see *Multiple Configuration Files* on how they are merged), then environment variables, and finally commandline options.

The following option packs are supported in the current version:

- `v1`: Added as of version 2.0.0 . Implies `--list-strat as-set` and `--disable-error-format`. This pack was added to help support legacy deployments of Degasolv. It should be noted that to achieve full compatibility with Degasolv version 1, the argument `--version-comparison maven` should be used as well as this option pack. It could not be included in the option pack due to complications with the version comparison option and its relationship to how the `--package-system` option is affected by it.
- `multi-version-mode`: Added as of version 1.7.0 . Implies `--conflict-strat inclusive`, `--resolve-strat fast`, and `--disable-alternatives`.
- `firstfound-version-mode`: Added as of version 1.7.0 . Implies `--conflict-strat prioritized`, `--resolve-strat fast`, and `--disable-alternatives`.

Print the Help Page

Short option	`-h`
Long option	`--help`
Version introduced	1.0.2

`-h`, `--help`: Prints the help page. This can be used on every sub-command as well.

6.3 CLI for `display-config`

6.3.1 Usage Page for `display-config`

Running `degasolv display-config -h` returns a page that looks something like this:

```
Usage: degasolv <options> display-config <display-config-options>

Options are shown below. Default values are marked as <DEFAULT> and
  descriptions. Options marked with `**` may be
  used more than once.

      --search-directory DIR    .              Find degasolv cards here
      --index-file FILE         index.dsrepo   The name of the repo file
      --index-strat STRAT       priority       May be 'priority' or 'global'.
      --requirement REQ                        Resolve req. **
      --search-strat STRAT      breadth-first  May be 'breadth-first' or 'depth-first
↪'.
      --conflict-strat STRAT    exclusive      May be 'exclusive', 'inclusive' or
↪'prioritized'.
      --repository INDEX                       Search INDEX for packages. **
      --enable-alternatives                    Consider all alternatives (default)
      --id ID                                  ID (name) of the package
      --query QUERY                            Display packages matching query string.
      --disable-alternatives                   Consider only first alternatives
      --add-to INDEX                           Add to repo index INDEX
      --card-file FILE          ./out.dscard   The name of the card file
      --present-package PKG                    Hard present package. **
      --resolve-strat STRAT     thorough       May be 'fast' or 'thorough'.
      --location LOCATION                      URL or filepath of the package
      --package-system SYS      degasolv       May be 'degasolv' or 'apt'.
      --version-comparison CMP  semver         May be 'debian', 'maven', 'naive',
↪'python', 'rpm', 'rubygem', or 'semver'.
      --version VERSION                        Version of the package
  -h, --help                                   Print this help page
```

6.3.2 Overview of `display-config`

This subcommand introduced as of version 1.6.0.

The `display-config` command is used to print all the options in the *effective configuration*. It allows the user to debug configuration by printing the actual configuration used by Degasolv after all the command-line arguments and config files have been merged together. An example of this is found in the *config files section*.

As of version 1.6.0, `display-config` accepts any valid option in long form (`--long-form`) which is accepted by any other subcommand. This enables the user to print out the effective configuration resulting from multiple config files as well as any options that might be given on the CLI.

As of version 2.0.0, `display-config` honors the setting of `--output-format`, if given in the configuration or on the command line: It will output JSON if set to `json`, EDN if set to `edn` or what it printed before version 2.0.0 (pretty EDN) if set to `plain`.

6.4 CLI for `generate-card`

6.4.1 Usage Page for `generate-card`

Running `degasolv generate-card -h` returns a page that looks something like this:

```
Usage: degasolv <options> generate-card <generate-card-options>

Options are shown below. Default values are marked as <DEFAULT> and
  descriptions. Options marked with `**` may be
  used more than once.

  -C, --card-file FILE   ./out.dscard  The name of the card file
  -i, --id ID                          ID (name) of the package
  -l, --location LOCATION              URL or filepath of the package
  -m, --meta K=V                       Add additional metadata
  -r, --requirement REQ                List requirement **
  -v, --version VERSION                Version of the package
  -h, --help                           Print this help page

The following options are required for subcommand `generate-card`:

  - `-i`, `--id`, or the config file key `:id`.
  - `-v`, `--version`, or the config file key `:version`.
  - `-l`, `--location`, or the config file key `:location`.
```

6.4.2 Overview of `generate-card`

This subcommand introduced as of version 1.0.2.

This subcommand is used to generate a card file. This card file is used to represent a package within a Degasolv repository. It is placed in a directory with other card files, and then the `generate-repo-index` command is used to search that directory for card files to produce a repository index.

6.4.3 Explanation of Options for `generate-card`

Specify Location of the Card File

Short option	`-C FILE`
Long option	`--card-file FILE`
EDN config file key	`:card-file "FILE"`
JSON config file key	`"card-file": "FILE"`
Environment variable	`DEGASOLV_CARD_FILE="FILE"`
Version introduced	1.0.2

Specify the name of the card file to generate. It is best practice to name this file after the name of the file referred to by the package's location with a `.dscard` extension. For example, if I created a card using the option `--location http://example.com/repo/a-1.0.zip`, I would name the card file `a-1.0.zip.dscard`, as in `--card-file a-1.0.zip.dscard`. By default, the card file is named `out.dscard`.

Specify the ID (Name) of the Package

Short option	`-i ID`
Long option	`--id ID`
EDN config file key	`:id "ID"`
JSON config file key	`"id": "ID",`
Environment variable	`DEGASOLV_ID="ID"`
Version introduced	1.0.2

Required. Specify the ID of the package described in the card file. The ID serves both as a unique identifier for the package and its name. It may be composed of any characters other than the following characters: `<>=!,;|`.

Specify the Location of the Package

Short option	`-l LOCATION`
Long option	`--location LOCATION`
EDN config file key	`:location "LOCATION"`
JSON config file key	`"location": "LOCATION",`
Environment variable	`DEGASOLV_LOCATION="LOCATION"`
Version introduced	1.0.2

Required. Specify the location of the file associated with the package to be described in the generated card file. Degasolv does not place any restrictions on this string; it can be anything, including a file location or a URL.

Specify Additional Metadata for a Package

Short option	`-m K=V`
Long option	`--meta K=V`
EDN config file key	`:meta {:key1 "value1" ...}`
JSON config file key	`"meta": {"key1": "value1", ...},`
Environment variable	`DEGASOLV_META="k1=v1^k2=v2..."`
Version introduced	1.11.0

Specify additional metadata about the package within the card file. This metadata will stay with the package information in its card file. It will also be printed with other package information about the package when the package is printed after dependency resolution when *resolve-locations* subcommand is called, provided that the *output-format* option is also used in a mode other than `plain`.

This is a powerful feature allowing the operator to build tooling on top of Degasolv. For example, now the operator may store the sha256 sum of the artifact, the location of its PGP signature, a list of scripts useful in the build contained within the artifact, etc.

For key/value pairs specified on the command line, keys are turned into EDN keywords (e.g., `:K`) internally and values are simply taken as strings. Additional metadata can also be specified from a configuration file as well. When they are specified via config file, they may be any data type allowed by EDN.

Key/value pairs specified via configuration file must be children of the top-level `:meta` key, like this:

```
{
    ...
    :meta {
        :sha256sum "sumsumsum"
        :otherkey "suchvalue"
        :key3 ["values", "can", "be", "lists"]
        :key4 {:key1 "or",
               :key2 "maps"}
    }
}
```

If used from the config file, the map's keys and values will be placed directly in to the card file. If keys `:id`, `:version` `:location`, or `:requirements` are specified in the config file, or keys `id=`, `version=`, `location=`, or `requirements=` on the CLI, they will be ignored.

Specify a Requirement for a Package

Short option	`-r REQ`
Long option	`--requirement REQ`
EDN config file key	`:requirements ["REQ1", ...]`
JSON config file key	`"requirements": ["REQ1", ...],`
Environment variable	`DEGASOLV_REQUIREMENTS="r1^r2..."`
Version introduced	1.0.2

List a requirement (dependency) of the package in the card file. May be specified one or more times as a command line option, or once as a list of strings in a configuration file. See *Specifying a requirement* for more information.

Specify a Version for a Package

Short option	`-v VERSION`
Long option	`--version VERSION`
EDN config file key	`:version "VERSION"`
JSON config file key	`"version": "VERSION",`
Environment variable	`DEGASOLV_VERSION="VERSION"`
Version introduced	1.0.2

Required. Specify the name of the package described in the card file.

Print the `generate-card` Help Page

Short option	`-h`
Long option	`--help`
Version introduced	1.0.2

Print a help page for the subcommand `generate-card`.

6.5 CLI for `generate-repo-index`

6.5.1 Usage Page for `generate-repo-index`

Running `degasolv generate-repo-index -h` returns a page that looks something like this:

```
Usage: degasolv <options> generate-repo-index <generate-repo-index-options>

Options are shown below. Default values are marked as <DEFAULT> and
  descriptions. Options marked with `**` may be
  used more than once.

  -a, --add-to INDEX                          Add to repo index INDEX
  -d, --search-directory DIR    .             Find degasolv cards here
  -I, --index-file FILE         index.dsrepo  The name of the repo file
  -O, --index-sort-order ORDER  descending    May be 'ascending' or 'descending'.
  -V, --version-comparison CMP  maven         May be 'debian', 'maven', 'naive',
↪'python', 'rpm', 'rubygem', or 'semver'.
  -h, --help                                  Print this help page
```

6.5.2 Overview of `generate-repo-index`

This subcommand introduced as of version 1.0.2.

This subcommand is used to generate a repository index file. A repository index file lists all versions of all packages in a particular Degasolv repository, together with their locations. This file's location, whether by file path or URL, would then be given to `resolve-locations` and `query-repo` commands as Degasolv repositories.

6.5.3 Explanation of Options for `generate-repo-index`

Specify the Repo Search Directory

Short option	`-d DIR`
Long option	`--search-directory DIR`
EDN config file key	`:search-directory "DIR"`
JSON config file key	`"search-directory": "DIR",`
Environment variable	`DEGASOLV_SEARCH_DIRECTORY="DIR"`
Version introduced	1.0.2

Look for Degasolv card files in this directory. The directory will be recursively searched for files with the `.dscard` extension and their information will be added to the index. Default value is the present working directory (`.`).

Specify the Repo Index File

Short option	`-I FILE`
Long option	`--index-file FILE`
EDN config file key	`:index-file "FILE"`
JSON config file key	`"index-file": "FILE",`
Environment variable	`DEGASOLV_INDEX_FILE="FILE"`
Version introduced	1.0.2

Write the index file at the location `FILE`. Default value is `index.dsrepo`. It is good practice to use the default value.

Specify the Index Sort Order

Short option	`-O ORDER`
Long option	`--index-sort-order ORDER`
EDN config file key	`:index-sort-order "ORDER"`
JSON config file key	`"index-sort-order": "ORDER",`
Environment variable	`DEGASOLV_INDEX_SORT_ORDER="ORDER"`
Version introduced	2.1.0

Specify that the packages within the index should be sorted by version number in either `descending` or `ascending` order. This has a significant impact on which version Degasolv chooses during dependency resolution.

Degasolv "trusts" the index. The index lists versions packages under a particular package name in a particular order, and Degasolv tries packages according to the order found in the index. This means that if the list of available package versions for any particular package name are sorted in descending order by version, then Degasolv will try the latest versions first. This is almost always what admins want in most dependency settings, and so has been the default for Degasolv before version 2.1.0 .

However, with the advent of golang's use of Minimum Version Selection, a use case has arisen for picking the smallest version first as part of resolution.

As of version 2.1.0, specific performance enhancements (internally labelled "version suggestion"), together with the option to specify an `ascending` version index sort order, allows the admin to ask Degasolv to practice minimum version selection.

Specify the Version Comparison Algorithm

Short option	`-V CMP`
Long option	`--version-comparison CMP`
EDN config file key	`:version-comparison "CMP"`
JSON config file key	`"version-comparison": "CMP",`
Environment variable	`DEGASOLV_VERSION_COMPARISON="CMP"`
Version introduced	1.8.0

Use the specified version comparison algorithm when generating the repository index. When repository indexes are generated, lists of packages representing different versions of each named package are created within the index. These lists are sorted in descending order by version number, so that the latest version of a given package is tried first when resolving dependencies.

This option allows the operator to change what version comparison algorithm is used. May be `debian`, `maven`, `naive`, `python`, `npm`, `rubygem`, or `semver`. As of version 2.0, the default algorithm is `semver`.

> **Caution:** This is one of those options that should not be used unless the operator has a good reason, but it is available and usable if needed.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. Similar options are availabe for the `resolve-locations` subcommand and the `query-repo` subcommand. They should all

agree when used within the same site. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

Add to an Existing Repository Index

Short option	`-a INDEX`
Long option	`--add-to INDEX`
EDN config file key	`:add-to "INDEX"`
JSON config file key	`"add-to": "INDEX",`
Environment variable	`DEGASOLV_ADD_TO="INDEX"`
Version introduced	1.0.2

Add to the repository index file found at `INDEX`. In general, it is best to simply regenerate a new repository index fresh based on the card files found in a search directory; however, it may be useful to use this option to generate a repository file incrementally.

For example, a card file might be generated during a build, then added to a repository index file in the same build script:

```
#!/bin/sh

degasolv generate-card \
  -i "a" -v "1.0.0" -l "http://example.com/repo/a-1.0.0.zip" \
  -C "a-1.0.0.zip.dscard"

degasolv generate-repo-index \
  -I "new-index.dsrepo" -a "http://example.com/repo/index.dsrepo" \
  -d "."

rsync -av a-1.0.0.zip.dscard user@example.com:/var/www/repo/
rsync -av new-index.dsrepo user@example.com:/var/www/repo/index.dsrepo
```

In this example, a card file is generated. Then, a new repository is generated based on an existing index and a newly generated card file. Then it is copied up to the repo server, replacing the old index. The card file is copied up as well to preserve the record in the search directory on the actual repository server so that a repository index could be generated on the server in the usual way later.

`INDEX` may be a URL or a filepath. Both HTTP and HTTPS URLs are supported. As of version 1.3.0, an `INDEX` may be specified as `-`, the hyphen character. If `INDEX` is `-`, Degasolv will read standard input instead of any specific file or URL.

6.6 CLI for `resolve-locations`

6.6.1 Usage Page for `resolve-locations`

Running `degasolv resolve-locations -h` returns a page that looks something like this:

```
Usage: resolve-locations <options>

Options are shown below. Default values are listed with the
  descriptions. Options marked with `**` may be
```

(continues on next page)

(continued from previous page)

```
  used more than once.

  -a, --enable-alternatives                          Consider all alternatives␣
↪(default)
  -A, --disable-alternatives                         Consider only first alternatives
  -e, --search-strat STRAT           breadth-first   May be 'breadth-first' or 'depth-
↪first'.
  -g, --enable-error-format                          Enable output format for errors␣
↪(default)
  -G, --disable-error-format                         Disable output format for errors
  -f, --conflict-strat STRAT         exclusive       May be 'exclusive', 'inclusive'␣
↪or 'prioritized'.
  -L, --list-strat STRAT             lazy            May be 'as-set', 'lazy' or 'eager
↪'.
  -o, --output-format FORMAT         plain           May be 'plain', 'edn' or 'json'
  -p, --present-package PKG                          Hard present package. **
  -r, --requirement REQ                              Resolve req. **
  -R, --repository INDEX                             Search INDEX for packages. **
  -s, --resolve-strat STRAT          thorough        May be 'fast' or 'thorough'.
  -S, --index-strat STRAT            priority        May be 'priority' or 'global'.
  -t, --package-system SYS           degasolv        May be 'degasolv', 'apt', or
↪'subproc'.
  -u, --subproc-output-format FORMAT  json           Whether to read `edn` or `json`␣
↪from the exe's output
  -V, --version-comparison CMP       maven           May be 'debian', 'maven', 'naive
↪', 'python', 'rpm', 'rubygem', or 'semver'.
  -x, --subproc-exe PATH                             Path to the executable to call␣
↪to get package data
  -h, --help                                         Print this help page

The following options are required:

  - `-R`, `--repository`, or the config file key `:repositories`.
  - `-r`, `--requirement`, or the config file key `:requirements`.
```

6.6.2 Overview of `resolve-locations`

This subcommand introduced as of version 1.0.2.

The `resolve-locations` command searches one or more repository index files, and uses the package information in them to attempt to resolve the requirements given at the command line. If successful, it exits with a return code of 0 and outputs the name of each package in the solution it has found, together with that package's location.

If the command fails because of dependency resolution problems, an exit code of 3 is returned. The output from such a run might look like this:

```
The resolver encountered the following problems:

Clause: e>=1.1.0,<2.0.0
- Packages selected:
  - b==2.3.0 @ https://example.com/repo/b-2.3.0.zip
  - d==0.8.0 @ https://example.com/repo/d-0.8.0.zip
- Packages already present:
  - x==0.1.0 @ already present
  - y==0.2.0 @ already present
```

(continues on next page)

(continued from previous page)

```
- Alternative being considered: e>=1.1.0,<2.0.0
- Package in question was found in the repository, but cannot be used.
- Package ID in question: e
```

As shown above, a list of clauses is printed. Each clause is an alternative (part of a requirement) that the resolver could not fulfill or resolve. Each field is explained as follows:

1. `Packages selected`: This is a list of packages found in order to resolve previous requirements before the "problem" clause was encountered.
2. `Packages already present`: Packages which were given to Degasolv using the *present package* option. If none were specified, this will show as `None`.
3. `Alternative being considered`: This field displays what alternative from the requirement was being currently considered when the problem was encountered.
4. The next field gives a reason for the problem.
5. `Package ID in question`: This field displays the package searched for when the problem was encountered.

6.6.3 Explanation of Options for `resolve-locations`

Enable the Use of Alternatives

Short option	`-a`
Long option	`--enable-alternatives`
EDN config file key	`:alternatives true`
JSON config file key	`"alternatives": true,`
Environment variable	`DEGASOLV_ALTERNATIVES="true"`
Version introduced	1.5.0

Consider all *alternatives* encountered while resolving dependencies. This is the default behavior. It allows the developers and packagers to decide whether or not to use alternatives. As alternatives are generally expensive to resolve, packagers should of course use them with caution. If this option occurs together with the `--disable-alternatives` option on a command line, the last argument of the two specified wins.

Disable the Use of Alternatives

Short option	`-A`
Long option	`--disable-alternatives`
EDN config file key	`:alternatives false`
JSON config file key	`"alternatives": false,`
Environment variable	`DEGASOLV_ALTERNATIVES="false"`
Version introduced	1.5.0

Consider only the first of any given set of *alternatives* for any particular requirement while resolving dependencies. It allows the package consumer to debug dependency resolution issues. This is especially useful when alternatives are used frequently in specifying requirements by packagers, thus causing performance issues on the part of the package consumers; or, when trying to figure out why dependencies won't resolve properly. If this option occurs together with the `--enable-alternatives` option on a command line, the last argument of the two specified wins.

Note: Use of this option defeats the purpose of Degasolv supporting alternatives in the first place. This option is intended generally for use when debugging a build. If it *is* used routinely, it should be used *site-wide*.

Specify Solution Search Strategy

Short option	`-e STRAT`
Long option	`--search-strat STRAT`
EDN config file key	`:search-strat "STRAT"`
JSON config file key	`"search-strat": "STRAT",`
Environment variable	`DEGASOLV_SEARCH_STRAT="STRAT"`
Version introduced	1.8.0

This option determines whether breadth first search or depth first search is used during package resolution. Valid values are `depth-first` to specify depth-first search or `breadth-first` to specify breadth-first search. This option is set to `breadth-first` by default.

Specify Conflict Strategy

Short option	`-f STRAT`
Long option	`--conflict-strat STRAT`
EDN config file key	`:conflict-strat "STRAT"`
JSON config file key	`"conflict-strat": "STRAT",`
Environment variable	`DEGASOLV_CONFLICT_STRAT="STRAT"`
Version introduced	1.1.0

This option determines how encountered version conflicts will be handled. Valid values are `exclusive`, `inclusive`, and `prioritized`. The default setting is `exclusive` and this setting should work for most environments.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

- If set to `exclusive`, all dependencies and their version specifications must be satisfied in order for the command to succeed, and only one version of each package is allowed. This is the default option, and is the safest, though it may carry with it significant performance ramifications. It turns dependency resolution into an NP hard problem. This is normally not a problem since the number of dependencies at most organizations (on the order of hundreds) is relatively small, but it is something of which the reader should be aware.
- If set to `inclusive`, all dependencies and their version specifications must be satisfied in order for the command to succeed, but multiple versions of each package are allowed to be part of the solution. To call for similar behavior to ruby's gem or node's npm, for example, set `--conflict-strat` to `inclusive` and set `--resolve-strat` to `fast`. This can be easily and cleanly specified done by using the `multi-version-mode` *option pack*.
- If set to `prioritized`, then the first time a package is required and is found at a particular version, it will be considered to fulfill the all other encountered requirements asking for that package. This is intended to mimic the behavior of java's maven package manager.

It means that, for example, if package `a` at version `1` requires package `b` at version `1` and also package `c` at version `1`; and package `c` at version `1` requires package `b` at version `2`; then the packages `a` at version `1`, the package `b` at version `1`, and the package `c` at version `1` will be found. Despite the fact that `c` needed `b` to be at version `2`, it had already been found at version `1` and that version was assumed to fulfill all requirements asking for package `b`.

To mimic the behavior of maven, set `--conflict-strat` to `prioritized` and `--resolve-strat` to `fast`. This can be easily and cleanly specified done by using the `firstfound-version-mode` *option pack*.

Specify List Strategy

Short option	`-L STRAT`
Long option	`--list-strat STRAT`
EDN config file key	`:list-strat "STRAT"`
JSON config file key	`"list-strat": "STRAT",`
Environment variable	`DEGASOLV_LIST_STRAT="STRAT"`
Version introduced	1.12.0

This option determines how packages will be listed once they are resolved. Valid values are `as-set`, `lazy`, and `eager`. As of version 2.0.0, the default value is `lazy`.

When the value is `as-set`, packages are listed in no particular order.

When the value is `lazy` or `eager`, packages are listed according to the following rules:

1. Barring cases of circular dependency, the child dependencies of any package are always listed before the package they depend on.
2. Circular dependencies are handled properly, but which dependency comes first is not guaranteed in all cases. In these cases the resolver must choose which dependency to ignore when it sees both. It choses to ignore the "deeper" dependency rather then the "shallower" package in the package resolution graph. So, for example, if package `a` relies on package `b` and package `b` relies on package `a`, but `a` is encountered first, the dependency from `a` to `b` will be honored but the dependency from `b` to `a` will be ignored when deciding in what order to list packages.
3. Otherwise, dependee packages will be listed in the order that the requirements they fulfill are listed. This means that, all things being equal, a package resolving one requirement of a parent package will be printed before a package resolving a different requirement of a different package listed further down in the requirements list for the parent package.

   For example, if a Degasolv card file called "steel" is made using the below config file:

```
{
    :requirements ["wool", "wood", "sheep"]
}
```

   When resolved, the represented package would be printed (or appear in the `json` or `edn` output, if *output-format* is set) in this order:

```
wool==1.0 @ http://example.com/repo/wool-1.0.zip
wood==1.0 @ http://example.com/repo/wood-1.0.zip
sheep==1.0 @ http://example.com/repo/sheep-1.0.zip
steel==1.0 @ http://example.com/repo/steel-1.0.zip
```

   It is worth noting that command line arguments are listed in reverse order. Thus, generating a card file with arguments `-r wool -r wood -r sheep` would yield a list that looks like this:

```
sheep==1.0 @ http://example.com/repo/sheep-1.0.zip
wood==1.0 @ http://example.com/repo/wood-1.0.zip
wool==1.0 @ http://example.com/repo/wool-1.0.zip
steel==1.0 @ http://example.com/repo/steel-1.0.zip
```

The difference between these options is that `lazy` will list dependencies as late as possible while following the above rules, while a value of `eager` tells Degasolv to list dependencies as early as possible while following the above rules.

Enable Error Output Format

Short option	`-g`
Long option	`--enable-error-format`
EDN config file key	`:error-format true`
JSON config file key	`"error-format": true,`
Environment variable	`DEGASOLV_ERROR_FORMAT="true"`
Version introduced	1.12.0

This option extends the functionality of *output-format* to include when errors happen as well.

Normally, when the *output-format* key is specified, such as to cause Degasolv to emit JSON or EDN, this only happens if the command runs successfully. If package resolution was unsuccessful, an error message is printed to standard error and the program exits with non-zero return code. If `error-format` is specified, then any error information will be printed in the form of whatever *output-format* specifies to standard output, while still maintaining the same exit code.

When error information is returned via JSON or EDN, the keys are the same in the dictionary, except:

- The `result` key now has the value of `unsuccessful`.
- The `packages` key is not present.
- A new key, `problems`, appears in place of the `packages` key containing information describing what went wrong.

As of version 2.0, the default behavior is to have `:error-format` enabled.

Disable Error Output Format

Short option	`-G`
Long option	`--disable-error-format`
EDN config file key	`:error-format false`
JSON config file key	`"error-format": false,`
Environment variable	`DEGASOLV_ERROR_FORMAT="false"`
Version introduced	1.12.0

This option sets the `:error-format` flag to `false`.

Specify Output Format

Short option	`-o FORMAT`
Long option	`--output-format FORMAT`
EDN config file key	`:output-format "FORMAT"`
JSON config file key	`"output-format": "FORMAT",`
Environment variable	`DEGASOLV_OUTPUT_FORMAT="FORMAT"`
Version introduced	1.10.0; EDN introduced 1.11.0

Specify an output format. May be `plain`, `edn` or `json`. This output format only takes effect when the package resolution was successful.

The default output format is `plain`. It is a simple text format that was designed for ease of use within bash scripts while also being somewhat pleasant to look at.

Example output on a successful run when the format is set to `plain`:

```
c==3.5.0 @ https://example.com/repo/c-3.5.0.zip
d==0.8.0 @ https://example.com/repo/d-0.8.0.zip
e==1.8.0 @ https://example.com/repo/e-1.8.0.zip
b==2.3.0 @ https://example.com/repo/b-2.3.0.zip
```

In the above example out, each line takes the form:

```
<id>==<version> @ <location>
```

When the output format is JSON, the output would spit out a JSON document containing lots of different keys and values representing some of the internal state Degasolv had when it resolved the packages. Among those keys will be a key called "packages", and it will look something like this:

```
{
  "command": "degasolv",
  "subcommand": "resolve-locations",
  "options": {
    "requirements": [
      "b"
    ],
    "resolve-strat": "thorough",
    "index-strat": "priority",
    "conflict-strat": "exclusive",
    "search-directory": ".",
    "package-system": "degasolv",
    "output-format": "json",
    "version-comparison": "maven",
    "index-file": "index.dsrepo",
    "repositories": [
      "./index.dsrepo"
    ],
    "search-strat": "breadth-first",
    "alternatives": true,
    "present-packages": [
      "x==0.9.0",
      "e==1.8.0"
    ],
    "card-file": "./out.dscard"
  },
```

(continues on next page)

(continued from previous page)


```
"result": "successful",
"packages": [
  {
    "id": "d",
    "version": "0.8.0",
    "location": "https://example.com/repo/d-0.8.0.zip",
    "requirements": [
      [
        {
          "status": "present",
          "id": "e",
          "spec": [
            [
              {
                "relation": "greater-equal",
                "version": "1.1.0"
              },
              {
                "relation": "less-than",
                "version": "2.0.0"
              }
            ]
          ]
        }
      ]
    ]
  },
  {
    "id": "c",
    "version": "3.5.0",
    "location": "https://example.com/repo/c-3.5.0.zip",
    "requirements": []
  },
  {
    "id": "b",
    "version": "2.3.0",
    "location": "https://example.com/repo/b-2.3.0.zip",
    "requirements": [
      [
        {
          "status": "present",
          "id": "c",
          "spec": [
            [
              {
                "relation": "greater-equal",
                "version": "3.5.0"
              }
            ]
          ]
        }
      ],
      [
        {
          "status": "present",
          "id": "d",
          "spec": null
```


(continues on next page)

(continued from previous page)

```
          }
        ]
      ]
    }
  ]
}
```

If the output format is EDN, the output will be similar, except it will use the EDN format:

```
{
  :command "degasolv",
  :subcommand "resolve-locations",
  :options {
    :requirements ("a<=1.0.0"),
    :resolve-strat "thorough",
    :index-strat "priority",
    :conflict-strat "exclusive",
    :search-directory ".",
    :package-system "degasolv",
    :output-format "edn",
    :version-comparison "maven",
    :index-file "index.dsrepo",
    :repositories (
      "./index.dsrepo"
    ),
    :search-strat "breadth-first",
    :alternatives true,
    :card-file "./out.dscard"
  },
  :result :successful,
  :packages #{
    #degasolv.resolver/PackageInfo {
      :id "b",
      :version "2.3.0",
      :location "https://example.com/repo/b-2.3.0.zip",
      :requirements []
    },
    #degasolv.resolver/PackageInfo {
      :id "a",
      :version "1.0.0",
      :location "https://example.com/repo/a-1.0.0.zip",
      :requirements [
        [
          #degasolv.resolver/Requirement {
            :status :present,
            :id "b",
            :spec nil
          }
        ]
      ]
    }
  }
}
```

The output, if the format is not `plain`, will have the following top-level keys in it:

- `command`: This is will be `degasolv`.

- `subcommand`: This will reflect what subcommand was specified. In the current version, this will always be `resolve-locations`.
- `options`: This shows what options were given when Degasolv was run. Its contents should roughly reflect the output of `display-config` when run with similar options.
- `result`: This displays whether the run was successful or not. Since unsuccessful runs result in a printed error and not outputted JSON, this will be `successful`. At present, to determine whether a run was successful, use the return code of Degasolv rather than this key.
- `packages`: This displays the list of packages and, if present, any additional *meta-data* associated with the package.

Specify that a Package is Already Present

Short option	`-p PKG`
Long option	`--present-package PKG`
EDN config file key	`:present-packages ["PKG1", ...]`
JSON config file key	`"present-packages": ["PKG1", ...],`
Environment variable	`DEGASOLV_PRESENT_PACKAGES="P1^..."`
Version introduced	1.4.0

Specify a "hard present package". Specify `PKG` as `<id>==<vers>`, as in this example: `garfield==1.0`.

Doing this tells Degasolv that a package "already exists" at a particular version in the system or build, whatever that means. This means that when Degasolv encounters a requirement for this package, it will assume the package is already found and it will mark the dependency as resolved. On the other hand, Degasolv will not try to change or update the found package. If the version of the present package conflicts with requirements encountered, resolution of those requirements may fail.

This is another one of those options that is provided and, if needed, is meant to benefit the user; however, judicious use is recommended. If you don't know what you're doing, you probably don't want to use this option.

For example, if this option is used to tell Degasolv that, as part of a build, some packages have already been downloaded, Degasolv will not recommend that those packages be upgraded. This is the "hard" in "hard present package": If the user specifies via `--present-package` that a package is already found and usable, Degasolv won't try to find a new version for it; it assumes "you know what you're doing" and that the package(s) in question are not to be touched.

Specify a Requirement

Short option	`-r REQ`
Long option	`--requirement REQ`
EDN config file key	`:requirements ["REQ1", ...]`
JSON config file key	`"requirements": ["REQ1", ...],`
Environment variable	`DEGASOLV_REQUIREMENTS="R1^R2^..."`
Version introduced	1.0.2

Required. Resolve this requirement together with all other requirements given. May be specified one ore more times as a command line option, or once as a list of strings in a configuration file. See *Specifying a requirement* for more information.

The last requirement specified will be the first to be resolved. If the requirements are retrieved from the config file, they are resolved in order from first to last in the list. If requirements are specified both on the command line and in the configuration file, the requirements in the configuration file are ignored.

Specify a Repository to Search

Short option	`-R INDEX`
Long option	`--repository INDEX`
EDN config file key	`:repositories ["INDEX1", ...]`
JSON config file key	`"repositories": ["INDEX1", ...],`
Environment variable	`DEGASOLV_REPOSITORIES="I1^I2^..."`
Version introduced	1.0.2

Required. Search the repository index given by INDEX for packages when resolving the given requirements.

When the index strategy is `priority` The last repository index specified will be the first to be consulted. If the repository indices are retrieved from the config file, they are consulted in order from first to last in the list. If indices are specified both on the command line and in the configuration file, the indices in the configuration file are ignored. See *index strategy* for more information.

`INDEX` may be a URL or a filepath pointing to a **.dsrepo* file. For example, index might be *http://example.com/repo/index.dsrepo*. Both HTTP and HTTPS URLs are supported. As of version 1.1.0, If `INDEX` is `-` (the hyphen character), Degasolv will read standard input instead of any specific file or URL. Possible use cases for this include downloading the index repository first via some other tool (such as cURL). One reason users might do this is if authentication is required to download the index, as in this example:

```
curl --user username:password https://example.com/degasolv/index.dsrepo | \
    degasolv resolve-locations -R - "req"
```

Specify a Resolution Strategy

Short option	`-s STRAT`
Long option	`--resolve-strat STRAT`
EDN config file key	`:resolve-strat "STRAT"`
JSON config file key	`"resolve-strat": "STRAT",`
Environment variable	`DEGASOLV_RESOLVE_STRAT="I1^..."`
Version introduced	1.0.2

This option determines which versions of a given package id are considered when resolving the given requirements. If set to `fast`, only the first available version matching the first set of requirements on a particular package id is consulted, and it is hoped that this version will match all subsequent requirements constraining the versions of that id. If set to `thorough`, all available versions matching the requirements will be considered. The default setting is `thorough` and this setting should work for most environments.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

Specify an Index Strategy

Short option	`-S STRAT`
Long option	`--index-strat STRAT`
EDN config file key	`:index-strat "STRAT"`
JSON config file key	`"index-strat": "STRAT",`
Environment variable	`DEGASOLV_INDEX_STRAT="STRAT"`
Version introduced	1.0.2

Repositories are queried by package id in order to discover what packages are available to fulfill the given requirements. This option determines how multiple repository indexes are queried if there are more than one. If set to `priority`, the first repository that answers with a non-empty result is used, if any. Note that this is true even if the versions don't match what is required.

For example, if `<repo-x>` contains a package `a` at version `1.8`, and `<repo-y>` contains a package `a` at version `1.9`, then the following command wil fail:

```
java -jar ./degasolv-<version>-standalone.jar -R <repo-x> -R <repo-y> \
    -r "a==1.9"
```

While, on the other hand, this command will succeed:

```
java -jar ./degasolv-<version>-standalone.jar -R <repo-y> -R <repo-x> \
    -r "a==1.9"
```

By contrast, if `--index-strat` is given the STRAT of `global`, all versions from all repositories answering to a particular package id will be considered. So, both of the following commands would succeed, under the scenario presented above:

```
java -jar ./degasolv-<version>-standalone.jar -S global \
    -R <repo-x> -R <repo-y> -r "a==1.9"

java -jar ./degasolv-<version>-standalone.jar -S global \
    -R <repo-y> -R <repo-x> -r "a==1.9"
```

The default setting is `priority` and this setting should work for most environments.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

Specify a Package System

Short option	`-t SYS`
Long option	`--package-system SYS`
EDN config file key	`:package-system "SYS"`
JSON config file key	`"package-system": "SYS",`
Environment variable	`DEGASOLV_PACKAGE_SYSTEM="SYS"`
Version introduced	1.4.0

Specify package system to use. By default, this value is `degasolv`. This causes the Degasolv's `resolve-locations` command to behave normally.

Other available values are shown below.

The "apt" Package System

Experimental. The `apt` package system resolves using the APT debian package manager. When using this method, *specify repositories* using the format:

```
{binary-amd64|binary-i386} <url> <dist> <pool>
```

Or, in the case of naive apt repositories:

```
{binary-amd64|binary-i386} <url> <relative-path>
```

For example, I might use the repository option like this:

```
degasolv resolve-locations \
    -R "binary-amd64 https://example.com/ubuntu/ /"
    -t "apt" \
    --requirement "ubuntu-desktop"
```

Or this:

```
degasolv resolve-locations \
    -R "binary-amd64 https://example.com/ubuntu/ yakkety main" \
    -R "binary-i386 https://example.com/ubuntu/ yakkety main" \
    -t "apt" \
    --requirement "ubuntu-desktop"
```

Degasolv does not currently support APT dependencies between machine architectures, as in `python:i386`. Also, every Degasolv repo is currently architecture-specific; each repo has an associated architecture, even if that architecture is `any`.

The "subproc" Package System

The `subproc` package system allows the user to give Degasolv package information via a subprocess (shell-out) command. A path to an executable on the filesystem is given via the *subproc-exe* option. For each repository specified via the *repository option*, the subproc executable path is executed with the string given for the repository as its only argument. The executable is expected to print out JSON or EDN to standard output, depending on the value of the *subproc-output-format* option.

The output should be a dictionary of packages listed by name. The value for each dictionary key should be an array of dictionaries, with each dictionary giving information about a particular package instance. Within each package instance dictionary, there should exist the keys `id` for the package name, `version` for its version, and `location` giving its location. Any requirements for the package instance should be listed under the `requirements` key according to the rules laid out in *Specifying a requirement*.

This information will then be read into Degasolv and used to resolve dependencies.

If the format is JSON, which is the default, the output should be of the form:

```
{
    "pkgname": [
        {
            "id": "pkgname",
            "version": "p.k.g-version",
            "location": "pkg-url",
            "requirements": ["birch>=3.3", "lime|lemon"],
            <optional kv-pairs associated with package>
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    "otherpkgname": [...]
}
```

If the format is EDN, the output should be of the form:

```
{
    "pkgname" [
        # The following will be referred
        {
            :id "pkgname"
            :version "p.k.g-version"
            :location" "pkg-url"
            :requirements ["birch>=3.3" "lime|lemon"]
            <optional kv-pairs associated with package>
        }
    ]
    "otherpkgname" [...]
}
```

Any additional kv-pairs specified in a package's record as shown above will appear in the resolution output if the *output-format* option is set to something other than `plain`.

If the executable exits with a non-zero error status code, Degasolv will print an error message looking like the following and also exit with a non-zero status code:

```
Error while evaluating repositories: Executable
`<path-to-exe>` given argument
`<repository-string>` exited with non-zero status `1`.
```

The resolver will search for packages in the order given in the output of the executable. Unless you have a good reason not to, you should list packages under the name of the package in the data structure on standard out in version-descending order.

Specify Subproc Package System Output Format

Short option	`-u FORMAT`
Long option	`--subproc-output-format FORMAT`
EDN config file key	`:subproc-output-format "FORMAT"`
JSON config file key	`"subproc-output-format": "FORMAT",`
Environment variable	`DEGASOLV_SUBPROC_OUTPUT_FORMAT="F"`
Version introduced	1.12.0

This option only takes effect if the `subproc` choice was listed for the *package-system* option. It says whether the executable used by Degasolv to get information needed to resolve dependencies will come in the form of an EDN or a JSON document. This option is set to `json` by default. See *package-system* docs for more information.

Specify the Version Comparison Algorithm

Short option	`-V CMP`
Long option	`--version-comparison CMP`
EDN config file key	`:version-comparison "CMP"`
JSON config file key	`"version-comparison": "CMP",`
Environment variable	`DEGASOLV_VERSION_COMPARISON="CMP"`
Version introduced	1.8.0

Use the specified version comparison algorithm when resolving dependencies.

This option allows the operator to change what version comparison algorithm is used. By default, the algorithm is "maven". May be "debian", "maven", "naive", "python" (PEP 440), "rpm", "rubygem", or "semver" (2.0.0). Version comparison algorithms are taken from the Serovers library. Descriptions for these algorithms can be found in the Serovers docs.

Caution: This is one of those options that should not be used unless the operator has a good reason, but it is available and usable if needed.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. Similar options are availabe for the `generate-repo-index` subcommand and the `query-repo` subcommand. They should all agree when used within the same site. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

Specify Subproc Package System Executable

Short option	`-x PATH`
Long option	`--subproc-exe PATH`
EDN config file key	`:subproc-exe "PATH"`
JSON config file key	`"subproc-exe": "PATH",`
Environment variable	`DEGASOLV_SUBPROC_EXE="PATH"`
Version introduced	1.12.0

This option only takes effect if the `subproc` choice was listed for the *package-system* option; however, it is required if the `subproc` choice was listed. It lists the path to the executable to use to get resolution information. See *package-system* docs for more information.

6.7 CLI for `query-repo`

6.7.1 Usage Page for `query-repo`

Running `degasolv query-repo -h` returns a page that looks something like this:

```
Usage: degasolv <options> query-repo <query-repo-options>

Options are shown below. Default values are marked as <DEFAULT> and
```

(continues on next page)

(continued from previous page)

```
  descriptions. Options marked with `**` may be
  used more than once.

  -g, --enable-error-format              Enable output format for errors (default)
  -G, --disable-error-format             Disable output format for errors
  -q, --query QUERY                      Display packages matching query string.
  -R, --repository INDEX                 Search INDEX for packages. **
  -S, --index-strat STRAT       priority  May be 'priority' or 'global'.
  -t, --package-system SYS      degasolv  May be 'degasolv' or 'apt'.
  -V, --version-comparison CMP  maven     May be 'debian', 'maven', 'naive', 'python',
↪ 'rpm', 'rubygem', or 'semver'.
  -h, --help                              Print this help page

The following options are required for subcommand `query-repo`:

  - `-R`, `--repository`, or the config file key `:repositories`.
  - `-q`, `--query`, or the config file key `:query`.
```

6.7.2 Overview of `query-repo`

This subcommand introduced as of version 1.0.2.

This subcommand queries a repository index or indices for packages. This comand is intended to be useful or debugging dependency problems. If errors occur relative to finding packages in the repository, as opposed to errors occuring because incorrect arguments were given, a return code of 2 is returned to the calling program (likely a shell).

6.7.3 Explanation of Options for `query-repo`

Enable Error Output Format

Short option	`-g`
Long option	`--enable-error-format`
EDN config file key	`:error-format true`
JSON config file key	`"error-format": true,`
Environment variable	`DEGASOLV_ERROR_FORMAT="true"`
Version introduced	1.12.0

This option extends the functionality of *output-format* to include when errors happen as well.

Normally, when the *output-format* key is specified, such as to cause Degasolv to emit JSON or EDN, this only happens if the command runs successfully. If querying the repo was unsuccessful, an error message is printed to standard error and the program exits with non-zero return code. If `error-format` is enabled, then any error information will be printed in the form of whatever *output-format* specifies to standard output, while still maintaining the same exit code.

When error information is returned via JSON or EDN, the keys are the same in the dictionary, except:

- The `result` key now has the value of `unsuccessful`.
- The `packages` key is not present.
- A new key, `problems`, appears in place of the `packages` key containing information describing what went wrong.

As of version 2.0, the default behavior is to have `:error-format` enabled.

Disable Error Output Format

Short option	`-G`
Long option	`--disable-error-format`
EDN config file key	`:error-format false`
JSON config file key	`"error-format": false,`
Environment variable	`DEGASOLV_ERROR_FORMAT="false"`
Version introduced	1.12.0

This option sets the `:error-format` flag to `false`.

Specify Output Format

Short option	`-o FORMAT`
Long option	`--output-format FORMAT`
EDN config file key	`:output-format "FORMAT"`
JSON config file key	`"output-format": "FORMAT"`
Environment variable	`DEGASOLV_OUTPUT_FORMAT="FORMAT"`
Version introduced	1.11.0

Specify an output format. May be `plain`, `edn` or `json`. By default the output format is `plain`. This output format only takes effect when the query returns a non-empty set of results. This is exactly like the *output-format* option for *resolve-locations*, except that the `subcommand` field is new returned as `query-repo`.

Specify Query

Short option	`-q QUERY`
Long option	`--query QUERY`
Config file key	N/A
Version introduced	1.0.2

Required. Query repository index or indices for a package. Syntax is exactly the same as requirements except that only one alternative may be specified (that is, using the `|` character or specifying multiple package ids), and the requirement must specify a present package (no `!` character may be used either). See *Specifying a requirement* for more information.

Examples of valid queries:

- `"pkg"`
- `"pkg!=3.0.0"`

Examples if invalid queries:

- `"a|b"`
- `"!a"`

Specify a Repository to Search

Short option	`-R INDEX`
Long option	`--repository INDEX`
EDN config file key	`:repositories ["INDEX1", ...]`
JSON config file key	`"repositories": ["INDEX1", ...],`
Environment variable	`DEGASOLV_REPOSITORIES="I1^I2^..."`
Version introduced	1.0.2

Required This option works exactly the same as the *repository option* for the `resolve-locations` command, except that instead of using the repositories for resolving requirements, it uses them for simple index queries. See that option's explanation for more information.

Specify an Index Strategy

Short option	`-S STRAT`
Long option	`--index-strat STRAT`
EDN config file key	`:index-strat "STRAT"`
JSON config file key	`"index-strat": "STRAT",`
Environment variable	`DEGASOLV_INDEX_STRAT="STRAT"`
Version introduced	1.0.2

This option works exactly the same as the *index strategy* option for the `resolve-locations` command, except that it is used for simple index queries. See that option's explanation for more information.

Specify a Package System

Short option	`-t SYS`
Long option	`--package-system SYS`
EDN config file key	`:package-system "SYS"`
JSON config file key	`"package-system": "SYS",`
Environment variable	`DEGASOLV_PACKAGE_SYSTEM="SYS"`
Version introduced	1.4.0

This option works exactly the same as the *package system* option for the `resolve-locations` command, except that it is used for simple index queries. See that option's explanation for more information.

Specify the Version Comparison Algorithm

Short option	`-V CMP`
Long option	`--version-comparison CMP`
EDN config file key	`:version-comparison "CMP"`
JSON config file key	`"version-comparison": "CMP",`
Environment variable	`DEGASOLV_VERSION_COMPARISON="CMP"`
Version introduced	1.8.0

Use the specified version comparison algorithm when querying the repository.

This option allows the operator to change what version comparison algorithm is used. By default, the algorithm is "maven". May be "debian", "maven", "naive", "python" (PEP 440), "rpm", "rubygem", or "semver" (2.0.0). Version comparison algorithms are taken from the Serovers library. Descriptions for these algorithms can be found in the Serovers docs.

Caution: This is one of those options that should not be used unless the operator has a good reason, but it is available and usable if needed.

Note: This option should be used with care, since whatever setting is used will greatly alter behavior. Similar options are availabe for the `generate-repo-index` subcommand and the `resolve-locations` subcommand. They should all agree when used within the same site. It is therefore recommended that whichever setting is chosen should be used *site-wide* within an organization.

6.8 Specifying a requirement

Unless otherwise noted, features in this section were introduced as of version 1.0.2 or earlier.

A requirement is given as a string of text. A requirement consists of one or more *alternatives*. Any of the alternatives will satisfy the requirement. Alternatives are specified by a bar character (`|`), like this:

```
"<alt1>|<alt2>|<alt3>"
```

Or, more concretely:

```
"hickory|maple|oak"
```

Alternatives will be considered in order of appearance.

Caution: In general, specifying more than one alternative is mostly unecessary, and should generally be avoided. This is because specifying too many alternatives tends to impact performance significantly; but they are available and usable if needed.

Each alternative is composed of a package id and an optional specification of what versions of that package satisfy the alternative, like this:

```
"<pkgid><version spec>"
```

For example:

```
"hickory>=3.0"
```

A version spec is a boolean expression of version predicates describing what versions may satisfy the alternative. The character `;` represents discution (OR) and the character `,` represents conjunction (AND), like this:

```
"<pred1>,<pred2>;<pred3>,<pred4>"
```

This is interpreted as:

```
"(<pred1> AND <pred2>) OR (<pred3> AND <pred4>)"
```

For example, this expression:

```
"spruce>=1.0.0,<2.0.0;>=3.0.0,<4.0.0"
```

Is interpreted as:

```
"spruce at version ((>=1.0.0 AND <2.0.0) OR (>=3.0.0 AND <4.0.0))"
```

6.8.1 Comparison Operators

Each version predicate is composed of a comparison operator and a valid version against which to compare a package's version. The character sequences <, <=, !=, ==, >=, and > represent the comparisons "older than", "older than or equal to", "not equal to", "equal to", "newer than or equal to", and "newer than", respectively, using whatever version comparison algorithm was specified using the CLI, or using the maven version comparison algorithm by default.

In addition to the above operators, three other version spec operators are provided:

- The "matches" operator: <>. *Introduced of version 1.8.0.* This operator is given in a version spec as <>REGEX. The version of any package found during the resolution process must match the given java regular expression. Examples:
  - The expression <>\d+\.\d+\.\d+ matches any version containing a three-part version in it.
  - The expression <>f[ea]{2}ture matches any version containing the strings "feature", "faeture", "feeture" or "faature".
- The "in-range" operator: =>. *Introduced as of version 1.8.0.* This operator is given in a version spec as =>RANGE. The version of any package found during the resolution process must be in the given version range. Examples:
  - The expression =>3.x matches the versions 3.0.0, 3.0.0.0 and 3.0 but not 4.0 or higher.
  - The expression =>3.3.x matches the versions 3.3.0, 3.3.8 and 3.3.8.99999 but not 3.4.0.

  Ranges are calculated in the following way:
  - Any non-digit characters found on the end of the RANGE string are removed.
  - All digit characters found on the end of the RANGE string are converted into a number and incremented. The incremented number is then put back into the version string, replacing any digit characters that were at the end of the string before. So, 3.x becomes 4, 3. becomes 4, and 2ormore becomes 3.
  - Finally, any versions comparing greater than or equal to the original RANGE string, but less than the incremented version string as computed in the previous step, are considered for dependency resolution.
- The "pessimistic greater-than" operator: ><. *Introduced as of version 1.9.0.* This operator is given in a version spec as ><VERS. The version of any package found during the resolution process must be greater or equal to the given version but less than the next major version. Examples:
  - The expression ><3.2.1 matches the versions 3.2.1, 3.4.3 but not 4.0.0 or higher, nor does it match 3.2.0.
  - The expression ><3.3.3 matches the versions 3.3.3, 3.3.8 and 3.9.8 but not 4.0.0.

  "The next major version" is calculated similarly to how ranges are calculated:
  - The first found set of digit characters found in the VERS string are converted into a number and incremented. The remainder of the version string after the incremented number is discarded.
  - Any versions comparing greater than or equal to the original VERS string, but less this new "incremented" version string as computed in the previous step, are considered for dependency resolution.

6.8.2 Examples

The following are examples of valid alternatives, together with their english interpretations:

Alternative	English Interpretation
`"oak"`	Find package `oak`
`"pine>1.0"`	Find package `pine` of version newer than `1.0`
`"pine><3.4.1-alpha8"`	Find package `pine` of version newer than or equal to `3.4.1-alpha8` but less than `4`.
`"fir<>\\d+\\.8"`	Find package `fir` containing "<digits>.8" somewhere in the version string
`"cedar=>3.x"`	Find package `cedar` at version greater or equal to major component `3` but less than `4`
`"hickory>1.0,<=2.0"`	Find package `hickory` with version newer than''1.0'' and older than or equal to `2.0`.
`"fir<=2.0;>3.5,!=3.8"`	Find a package `fir` with version (newer than `1.0` and older than or equal to `2.0`) OR (with version newer than `3.5` but not equal to `3.8`)

Note: To make debugging easier, try to keep things as simple as possible. Try not to make requirement strings very long. When using the `inclusive` or `priority` *conflict strategies*, it is recommended to specify exact package names and versions, like this: `pkgname==1.0.0`. The simpler the requirement string, the easier it will be to untangle any untoward dependency problems.

Negative alternatives are requirements that all packages with a particular id and matching a particular version spec must be absent from the list of packages found when resolving dependencies. To negate an alternative, prepend it with the `!` character.

For example, the following alternative means "make sure the `spruce` package is not present in the list":

```
!spruce
```

This alternative means "If package a is present in the list, make sure its version is not in the range `(3.0,4.0]`":

```
!a>3.0,<=4.0
```

The following are practical examples of requirements, together with their interpretations.

Requirement	Explanation
`"oak|pine>5.0"`	Require `oak` at any version, or `pine` at versions greater than `5.0`
`"hickory>=3.0,<4.0"`	Require `hickory` at a `3.x` version.
`"!birch|birch<=3.0" "!birch>3.0"`	An important example. This demonstrates how to specify what maven calls a managed dependency. It means if `birch` is required by another package, ensure that its version is older than or equal to `3.0`. It is good practice to prefer the expression with only one alternative.
`"!oak|maple>3.0"`	If oak is installed, then make sure maple after version 3.0 is installed also.
`"oak|!pine"`	Require the presence of the `oak` package, or the absence of the `pine` package.

CHAPTER 7

Architecture

This article describes the overarching architecture of Degasolv, together with some explanation about some of the design decisions.

7.1 Background

At one of my previous jobs, I was a Build Engineer – a person who built the code that the developers wrote and made it available. I had *lots* of dependency problems coming at me from all different sides:

- One module in language A depending on another from language B
- Developers working with a language (at the time) with no clear package manager ecosystem (**cough** C++ **cough**)
- Package manager developers breaking builds with backwards-incompatible behavior
- A dependency graph that looked like a dream catcher

So I decided to build a tool that would do these things:

- Resolve dependencies the right way, safely
- Even resolve dependency chains for different package systems (apt, pip, java)
- Be super versatile and generic, able to be plugged into an arbitrary build script

7.2 Core Resolver

At the core of Degasolv is a monster method called resolve-dependencies. It is a rather large method with a backtracking SAT-solver-ish design. Originally it was written to have a *conflict-strat* of `exclusive` and a *resolve-strat* of `thorough` hard-coded. In other words, the "first class" original use case of Degasolv was a SAT-solver-class depedency resolver that only allowed a single version of any dependency, and ensured that all parties depending on that dependency had a chance to agree on what was chosen. These options were later added to allow Degasolv to act more like maven and give any Building Engineer using Degasolv useful "handbreaks" to change how resolution was being done in-house so that it could be modified to conform to business needs. Other options, such as *list-strat*<*list-strat*>

and *search-strat*<*search-strat*> were added as time progressed as well for similar reasons, and also, frankly, to fix bugs (behaviors that were never originally intended).

CHAPTER 8

Contributor Covenant Code of Conduct

8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at “djhaskin987 at gmail.com” . All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project’s leadership.

8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, version 1.4.

CHAPTER 9

Contributing Guide

Thank you so much for considering contribution to Degasolv!

First, please read the Degasolv *Contributor Covenant Code of Conduct*. This project will not take any contribution coming from those who do not abide by the code of conduct. This means that if a person is currently under disciplinary action via avenues set forth in that document, we will ignore your PR and/or any issues you may log.

A contribution can be large or small, code or non-code. To make a contribution, first log a GitHub issue. Talk about what you want, and ask for other's opinions.

When you go to make the PR, please use the following checklist to test whether or not it is likely to be accepted:

1. **Is it based on the ''develop'' branch?** Degasolv uses the git-flow framework for branch management. Please make PRs to the `develop` branch.
2. **Do you have tests in your PR, and do they pass?** Tests are in two places in Degasolv: the `test/degasolv` directory, where more or less normal unit tests reside; and the `test/resources/scripts` directory, where scripty-integration tests reside. You must have at least a script test (and preferrably one or more unit tests) as a "spot-check" of your feature if the PR is to be merged. The test need not be elaborate; a simple test is better than no tests.
3. **Is your PR backwards compatible?** The biggest feature Degasolv provides is backwards compatibility.

   We only consider backwards incompatible changes in the form of new options for the "correct" behavior and switching the default for that option in a new version, ideally a major version.

   If Degasolv breaks a build, it is a bug. If it breaks a build in a way that can't be fixed by configuration, it is a bad bug.

   A good test if a PR is "backwards compatible" is if 1) it changes any previously merged script test and 2) if it breaks any of them.
4. **Did you add documentation around the feature in your PR?** Generally this at least means adding something to the *Degasolv Command Reference* document.
5. **Did you add an entry to the Changelog?** This project keeps a curated *changelog*.

There are some exceptions to the above rules. For example, if your patch is less than two lines' difference from the previous version, your PR may be a "typo" PR, which may qualify to get around some of the above rules. Just ask the team on your GitHub issue.

CHAPTER 10

Roadmap

This file outlines what we plan on doing to democratize dependency management. It may or may not actually be implemented in the future, but represents a guide for contributors and users alike as to the hopes and vision for the future of the Degasolv developers.

10.1 Future Releases

- Tutorial-like help screens designed to keep people from needing to switch from docs to cli and back.
- Shortened versions of all subcommands, including documentation updates.
- Documentation and/or code on the topic of supporting the use case of different architectures of the same package using prioritized indexes of packages named the same with different contents.
- **Compile with GraalVM's ‘‘native-image‘‘**: Compile degasolv to machine code with GraalVM's `native-image` to decrease start-up times. This will likely coincide with upgrading to Clojure 1.11 because native-image doesn't work with Clojure 1.10.1 .

10.2 2.3.0

Firefighters need tools that can apply in many situations; similarly, ops and DevOps professionals, for whom we build this tool, need to a dependency management tool that can get them out of dependency hell no matter what their situation.

- The ability to slurp from JDBC URLs for indexes
  - An extension will be made to ensure that username and password can be specified along with a URL. Not all drivers support this and it is an important use case.
  - Generate-repo-index to support JDBC URLs
  - query-repo and resolve-locations to support JDBC URLs
  - If the database is empty or doesn't exist, it will be created on generate repo index or on index-add

- New subcommands: index-add, index-rm, to take away from and add to as in an installation/removal context
- New subcommand: resolve-dependents to find all dependents in an *index*
- USER GUIDES
  - How to use repositories as generic installation trackers
  - How to track dependencies between kubernetes services
  - How to track dependencies between cross-language builds and use this for that
  - Documentation and/or code on the topic of supporting the use case of different architectures of the same package using prioritized indexes of packages named the same with different contents.
  - How to use degasolv to manage a project installation for development purposes

CHAPTER 11

Authors and Contributions

We thank everyone that has contributed to Degasolv!

11.1 Authors

- Daniel Jay Haskin

11.2 Contributions

- Logo contributed to Degasolv by James Toney

CHAPTER 12

3rd Party Licenses

Degasolv is built on the shoulders of giants. Here are the licenses for the third party software that comes with Degasolv:

```
commons-io - 2.6 - Apache License, Version 2.0
clojure-complete - 0.2.5 - Eclipse Public License
org.apache.httpcomponents/httpmime - 4.5.8 - Apache License, Version 2.0
potemkin - 0.4.5 - MIT License
com.fasterxml.jackson.dataformat/jackson-dataformat-smile - 2.9.9 - The Apache␣
↪Software License, Version 2.0
cheshire - 5.9.0 - The MIT License
clj-http - 3.10.0 - The MIT License
org.apache.httpcomponents/httpclient - 4.5.8 - Apache License, Version 2.0
org.clojure/tools.cli - 0.3.5 - Eclipse Public License 1.0
com.fasterxml.jackson.dataformat/jackson-dataformat-cbor - 2.9.9 - The Apache␣
↪Software License, Version 2.0
org.flatland/ordered - 1.5.7 - Eclipse Public License - v 1.0
tigris - 0.1.1 - Eclipse Public License
clj-tuple - 0.2.2 - MIT License
org.apache.httpcomponents/httpcore - 4.4.11 - Apache License, Version 2.0
slingshot - 0.12.2 - Eclipse Public License 1.0
org.apache.httpcomponents/httpclient-cache - 4.5.8 - Apache License, Version 2.0
org.apache.httpcomponents/httpasyncclient - 4.1.4 - Apache License, Version 2.0
org.flatland/useful - 0.11.6 - Eclipse Public License - v 1.0
com.fasterxml.jackson.core/jackson-core - 2.9.9 - The Apache Software License,␣
↪Version 2.0
org.clojure/tools.reader - 0.7.2 - Eclipse Public License 1.0
org.clojure/clojure - 1.10.1 - Eclipse Public License 1.0
org.clojure/spec.alpha - 0.2.176 - Eclipse Public License 1.0
serovers - 1.6.2 - Eclipse Public License
riddley - 0.1.12 - MIT License
commons-logging - 1.2 - The Apache Software License, Version 2.0
org.apache.httpcomponents/httpcore-nio - 4.4.10 - Apache License, Version 2.0
org.clojure/tools.macro - 0.1.1 - Eclipse Public License 1.0
com.velisco/tagged - 0.5.0 - Eclipse Public License
org.clojure/core.specs.alpha - 0.2.44 - Eclipse Public License 1.0
commons-codec - 1.12 - Apache License, Version 2.0
```

CHAPTER 13

Indices and tables

- genindex
- modindex
- search