Common Lisp Project Manager

Eric Timmons
eric@timmons.dev

ABSTRACT

In this paper we describe the Common Lisp Project Manager (CLPM), a new addition to the Common Lisp dependency management ecosystem. CLPM provides a superset of features provided by the Quicklisp client, the current de facto project manager, while maintaining compatibility with both ASDF and the primary Quicklisp project distribution. These new features bring the Common Lisp dependency management experience closer to what can be achieved in other programming languages without mandating a particular development workflow/environment and without sacrificing Common Lisp's trademark interactive style of development.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software libraries and repositories*; *Application specific development environments.*

ACM Reference Format:
Eric Timmons. 2021. Common Lisp Project Manager. In *Proceedings of 14th European Lisp Symposium (ELS '21).* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

One way to manage the complexity of software is to reuse code by packaging up a project and using it as a dependency in another. Unfortunately, this ends up introducing another level of complexity: you have to manage your dependencies and their versions as well! In the Common Lisp world, this management is usually performed by three interacting pieces: ASDF [2] to ensure all dependencies are compiled and loaded in the right order and the version of each is sufficient, the Quicklisp client [6] for installing the dependencies locally and configuring ASDF to find them, and a Quicklisp distribution containing an index listing available projects and metadata about them.

While effective and widely used, this combination of components is missing many features that are taken for granted in other programming language specific package management ecosystems. In this paper we describe the Common Lisp Project Manager (CLPM)[1], a potential replacement for the Quicklisp client in the Common Lisp dependency management ecosystem that provides many of these missing features.

[1]CLPM is in the process of being renamed from the Common Lisp Package Manager. While in most communities "package" typically refers to some installable unit of software, it unfortunately collides with the use of "package" to describe symbol namespaces in Common Lisp

We begin by highlighting several of the key features that CLPM implements, along with the overarching design philosophy. We then describe the high-level design of CLPM. Last, we provide information on how to obtain CLPM and several examples on how to use it.

2 FEATURES AND PHILOSOPHY

CLPM has a set of features that, while commonly found in language specific package managers for other languages, have not seen wide adoption in the Common Lisp community. The most important of these include:

(1) support for HTTPS,
(2) the ability to manage both global and project-local contexts,
(3) the ability to "lock" (or "pin", "freeze", etc.) dependencies to specific versions and replicate that setup exactly on another machine,
(4) a robust CLI for easy interfacing with shell scripts and other languages,
(5) and the ability to install unreleased, development versions of dependencies from source control.

While CLPM is not the first to implement any of these given features for Common Lisp, we believe it is the first to do so in a complete package that also places minimal constraints on the workflow of the developer using it. For example, the quicklisp-https [11] project adds HTTPS support to the Quicklisp client, but it does so at the cost of requiring that Dexador [7] (and all of its dependencies, including foreign SSL libraries) be loaded into the development image.

Another example is the qlot [8] project. It adds project-local contexts (but not global), locking, a CLI, and using development versions of dependencies. One of the biggest drawbacks of qlot, however, is that its CLI requires Roswell [12], which in turn implies that Roswell is responsible for managing and installing your Lisp implementations.

As just alluded, the most important guiding principle of CLPM is that it should place minimal constraints on developer workflow. An example of this principle in action is illustrated by how CLPM is distributed. For Linux systems, a static executable is provided that runs on a wide variety of distributions with no dependencies. For MacOS systems, a binary is provided that requires only OS provided foreign libraries. And on Windows, CLPM is distributed using a standalone installer and again depends only on OS provided foreign libraries.

Perhaps the second most important principle is that CLPM should be highly configurable, yet provide a set of sane and safe defaults. A concrete example of this is shown by CLPM's behavior when installing or updating projects. By default, CLPM will describe the change about to be performed and require explicit user consent before making the modification. However, this behavior can be changed for developers that like to live on the edge or otherwise have complete trust in CLPM and the projects they are installing.

3 DESIGN

In this section we discuss the design of CLPM. First, we describe the overall architecture, including the three main CLPM components. Second, we describe some of the benefits the architecture provides and how CLPM leverages them. Third, we describe where CLPM locates the metadata for projects that it installs. Last, we describe global and project-local contexts in CLPM.

3.1 Architecture

CLPM is split into two user facing components: the worker and the client, as well as one internal component: the groveler. The worker is a standalone executable that is responsible for all the heavy lifting. The worker interacts with the network to download releases and metadata, performs dependency resolution, unpacks archives, and manages contexts. The worker is implemented in Common Lisp and distributed both as a precompiled executable (static executable for Linux) and source code for those that want to compile it themselves. It exposes both a CLI interface and a REPL interface. The CLI interface allows for easy integration with tools such as shell scripts and continuous integration services. The REPL interface is used primarily by the CLPM client.

The client is a small system, written in portable Common Lisp, with ASDF/UIOP as its only dependency. The client is meant to be loaded into a Lisp image to facilitate the interactive management of contexts and development of code. It does this by exposing a set of functions corresponding to the operations the worker can perform as well as integrating with ASDF's system search functions. Additionally, it has functionality to remove itself from an image if it is no longer required (such as when dumping an executable). In order to interact with the worker, the client spawns a new worker process, starts its REPL, and they communicate back and forth with S-expressions.

The last component is the groveler. Ideally users never interact directly with this component; instead, it is used by the worker to gather information from ASDF system definitions in a clean environment. Nominally, the data that CLPM require about a system would be distributed in metadata published by project indices. However this metadata is not available for development versions of a project. As ASDF allows systems to specify that some dependencies must be loaded before the system definition can even be processed, it is in general not possible to extract system metadata without running arbitrary code.

The groveler consists of a small set of portable Common Lisp functions that the worker loads into a fresh Common Lisp instance. Once loaded, the groveler and worker communicate via S-expressions, with the worker specifying which systems to load and extract information from and the groveler reporting back the information as well as any errors. If there is an error, the worker addresses it by recording any missing dependencies in the dependency resolution process, loading them into the groveler, and trying again. The worker keeps track of the project releases loaded in the groveler and starts a new groveler process if needed (e.g., the groveler has v2 of system A loaded, but a requirement determined later in the resolution process caps system A at v1).

3.2 Dependencies and Non-portable Code in the Worker

One benefit of the worker/client split is that it enables the worker to freely leverage both dependencies (Common Lisp and foreign) and non-portable implementation features.

There are two benefits with respect to dependencies in the worker. First, the worker can reuse code without worrying about interference with the code the user wishes to develop. This interference may manifest itself in many ways, including as package nickname conflicts, version incompatibilities (CLPM needs version x of a system, but the user's code needs version y), or image size (e.g., the user's code uses few dependencies and they care about the final size of the image for application delivery).

Second, the worker can use foreign libraries that it is unlikely the user needs loaded for their development. A prime example of this is libssl, which CLPM uses on Linux and MacOS systems to provide HTTP support. A second-order benefit of this approach is that CLPM can statically link foreign libraries into the worker so that the user does not even need to have them installed locally (if they install a pre-compiled version of CLPM at least).

The ability for the worker to freely use dependencies has an additional knock-on effect: the development of CLPM can help improve the state of Common Lisp libraries at large. To date, the development of CLPM has resulted in at least ten issues and merge requests being reported to upstream maintainers, eight of which have been merged or otherwise addressed. Additionally, it has been the motivating factor behind several contributions to SBCL to improve musl libc support.

The worker routinely needs to perform operations that require functionality beyond the Common Lisp specification. For example: loading foreign libraries, interfacing with the OS to set file time stamps when unpacking archives, and network communication to download code. While portability libraries exist for many of these features, they are not perfect and do not necessarily extend support to all implementations [9]. The worker/client split allows us to choose a small number of target implementations and focus our testing and distribution efforts using only those implementations while not worrying about restricting what implementation the user chooses to develop their code.

Currently, CLPM targets only SBCL [3] for worker support. This is due to SBCL's broad compatibility with OSes and CPU architectures. We look to extend worker support to at least one other implementation before CLPM reaches v1.0. The next implementation targeted by the worker will likely be ABCL [1] due to the ubiquity of the Java Virtual Machine.

3.3 Indices

Project indices are used to advertise projects and metadata about those projects, such as their released versions, what systems are available in each release, and the version number and dependencies of those systems. The most widely used project index in the Common Lisp community is the primary Quicklisp distribution. CLPM is tested against this index and has full support for interacting with it.

While there are other Quicklisp-like project indexes in the wild that work with the Quicklisp client, such as Ultralisp [5], CLPM may not work with all of them. This incompatibility is largely due to a lack of formal specification as to what constitutes a Quicklisp distribution, including what files are required and their contents. For instance, at the time of writing CLPM does not work with the Ultralisp distribution because Ultralisp does not publish a list of all its historical versions, unlike the primary Quicklisp distribution.

In addition to supporting Quicklisp distributions as project indices, CLPM supports Common Lisp Project Indices (CLPI). CLPI is part of the overarching CLPM project and seeks to fully document an index format as well as add features that CLPM can take advantage of that are missing from Quicklisp distributions. One feature CLPI adds is that system versions (taken from ASDF system definitions) are provided. Quicklisp provides only the date at which the project snapshot was taken.

A second major difference is that projects and systems are the top-level concepts in CLPI instead of distribution version. Additionally, CLPI defines the majority of its files to be append-only. These properties allow CLPM to be more efficient in terms of network usage as only the metadata for projects potentially needed in the context are transferred and metadata on new releases can be fetched incrementally over time.

3.4 Contexts

CLPM manages both global and project-local contexts. A context is defined as a set of project indices in use, a set of constraints describing the projects (and their versions) that should be available in the context, and a set of project releases that satisfy both the explicit constraints and the implicit constraints added by every project in the context (i.e., transitive dependencies).

Each global context is named by a string. CLPM provides tools that generate ASDF source registry configuration files so that you can add these global contexts to your default registry and have access to all projects installed in them without the need for CLPM at all (after installation, at least).

A project-local context (also known as a *bundle*) is named by a pathname that points to a file containing the first two elements of a context (the indices and constraints). After this context is installed, all of the context information is located in a file next to the file that names the context. The names of these files are typically `clpmfile` and `clpmfile.lock`. Both of these files are designed to be checked into a project's source control and contain enough information to reproduce the context on another machine.

Both CLPM's CLI and the client provide commands to add new constraints to a context, update an entire context so that every project is at the latest release that satisfies all constraints, or update just a subset of the projects in the context. Additionally, the CLI provides commands that can execute other, arbitrary, commands in an environment where ASDF is configured to find only the projects installed in the desired context.

Global contexts are largely inspired by Python virtual environments created by the virtualenvwrapper project [10]. Project-local contexts are largely inspired by Ruby's Bundler project [4].

4 USE

In this section we provide some examples of how to use CLPM. We focus on project-local contexts (bundles) as we believe these are a more broadly useful feature than global contexts.

4.1 Installing CLPM

Tarballs (and MSI installers) of the most recent CLPM release, along with up to date documentation, can be found at https://www.clpm.dev. Windows users merely need to run the installer. Linux and MacOS users need to only unpack the tarball in an appropriate location (typically `/usr/local/`). If you wish to install from source, the CLPM source code (as well as its issue tracker) can be found on the Common Lisp Foundation Gitlab instance at https://gitlab.common-lisp.net/clpm/clpm.

After installing CLPM, it is recommended that you configure ASDF to find the CLPM client. To do this, simply run `clpm client source-registry.d` and follow the instructions printed to the screen.

Last, you may wish to consider loading the CLPM client every time you start your Lisp implementation. You can determine the currently recommended way of doing so by running `clpm client rc` and reading the printed instructions. The remainder of this section assumes you have CLPM installed and have a REPL to an image where the CLPM client is loaded.

4.2 Creating a Bundle

The simplest possible bundle is one that uses a single project index and specifies that all constraints come from an .asd file. To create such a bundle you need to perform two actions. First, a `clpmfile` must be created. This can be done at the REPL using the client's `bundle-init` function. After running this function, you will have a `clpmfile` that looks similar to the following:

```
(:api-version "0.3")

(:source "quicklisp"
 :url "https://beta.quicklisp.org/dist/quicklisp.txt"
 :type :quicklisp)

(:asd "my-system.asd")
```

Notice that the first statement in the `clpmfile` declares the bundle API in use. This allows for the file format to evolve over time while maintaining backward compatibility. Second, notice that every directive is simply a plist. This makes it trivial for any Common Lisp project to read and manipulate the file.

After the `clpmfile` is created, the bundle must be installed (that is all the constraints must be resolved and dependencies downloaded). This can be done at the REPL using the `install` function.

CLPM does not believe in modifying a context without explicit permission from the developer. As such, the client will by default produce a condition before it performs any modifications. This condition describes the actions that are about to be performed and has two restarts established for it: one to approve and perform the changes, and one to abort. Therefore, upon evaluating `install`, you will be dropped into the debugger to approve the changes. Of course, this behavior can be customized.

4.3 Activating the Bundle

Once a bundle has been installed, the next step is to activate it. The activation process configures ASDF so that it can locate all systems in the bundle. The activation process also optionally integrates the CLPM client with ASDF so that CLPM is aware when you try to find a system that is not present in the bundle and can provide options on adding it to the bundle. This integration is enabled by default and discussed more next. Activation is performed using the `activate-context` function.

The CLPM CLI has an equivalent command that configures ASDF using only environment variables. This feature is particularly useful when running scripts or when running tests via CLI. For example, to run an SBCL process where ASDF is configured to use only the systems in the bundle *without* also needing to have the client loaded, simply run `clpm bundle exec -- sbcl` in the same directory as the `clpmfile`.

4.4 Modifying the Bundle

The most common modification made to a bundle is to add more dependencies to it. If your system acquires a new dependency you have two options on how to add it to the bundle. The first option is to explicitly reinstall the bundle using `install`. This option will find any new dependency and install it, while trying its best to not change the installed versions of the projects already in the bundle.

The second option is to just use `asdf:load-system` to reload your system. If you have the client's ASDF integration enabled, the client will notice that the system is missing from the bundle and take action. The default action is to signal a condition informing the developer of the situation with several restarts in place. Following CLPM's guiding principles, however, this behavior can be modified to, for example, automatically install the dependency à la the Quicklisp client.

5 CONCLUSION

We have presented CLPM — the Common Lisp Project Manager. CLPM introduces many features to the Common Lisp community that are taken for granted in other programming language specific package managers. Key among these features are HTTPS support, a standalone CLI to the worker, global and project-local context management, and lock files. Additionally, CLPM adds these features without forcing a particular development practice or environment and without sacrificing "Lispy-ness" or interactive development.

6 FUTURE WORK

We plan to continue developing CLPM and continue adding useful features. Two planned features of note are an extensible architecture for client/worker communication and the ability to add scripts from an installed project to a user's `PATH`. The former would enable a myriad of new configurations, including CLPM workers deployed as persistent daemons that communicate with clients over network sockets or setups that are Dockerized or otherwise isolated from other processes on the system.

In addition to improvements to CLPM itself, we aim to continue contributing back to the upstreams of our dependencies. Notably, we are in the process of interacting with the ASDF developers to add support for more expressive version numbers and dependency constraints in `defsystem` forms.

Last, it is our strong preference to enable developers to use the complete power of CLPM without introducing a split in the community with regards to project indices. Therefore, we would like to take lessons learned from and features added to CLPI and integrate them into Quicklisp distributions.

REFERENCES

[1] Armed Bear Common Lisp. https://abcl.org/.
[2] ASDF – Another System Definition Facility. https://common-lisp.net/project/asdf/.
[3] Steel Bank Common Lisp. http://www.sbcl.org/.
[4] André Arko and Engine Yard. Bundler: The best way to manage a Ruby application's gems. https://bundler.io/.
[5] Alexander Artemenko. Ultralisp - Fast Common Lisp Repository. https://ultralisp.org/.
[6] Zach Beane. Quicklisp. https://www.quicklisp.org/beta/.
[7] Eitaro Fukamachi. Dexador. https://github.com/fukamachi/dexador/, .
[8] Eitaro Fukamachi. Qlot. https://github.com/fukamachi/qlot/, .
[9] Nicolas Hafner. Common Lisp Portability Library Status. https://portability.cl/.
[10] Doug Hellmann. virtualenvwrapper. https://virtualenvwrapper.readthedocs.io/en/latest/.
[11] SANO Masatoshi. Quicklisp-HTTPS. https://github.com/snmsts/quicklisp-https/, .
[12] SANO Masatoshi. Roswell. https://github.com/roswell/roswell/, .