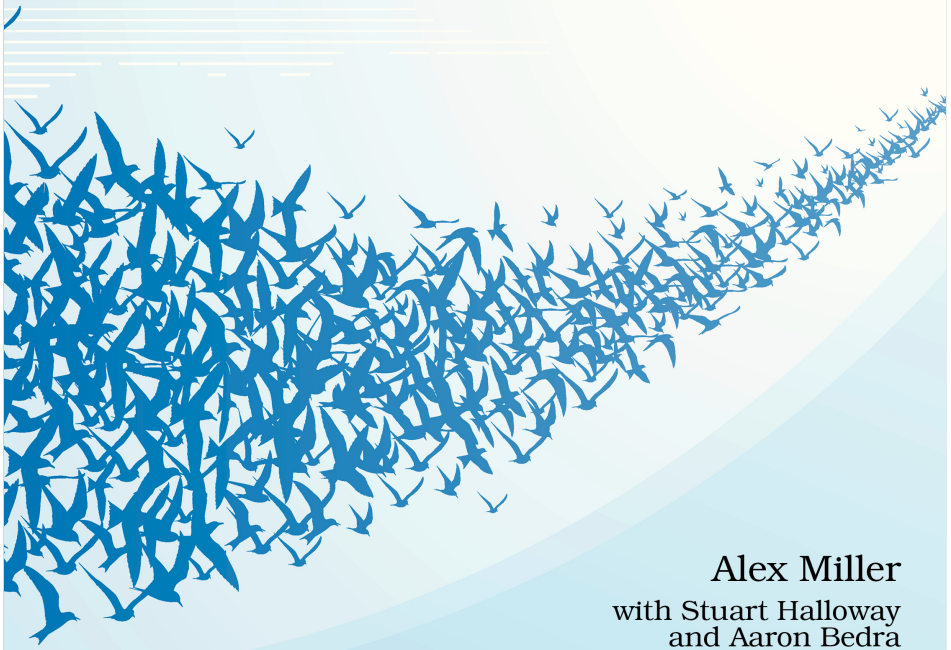
The Pragmatic Programmers

Programming Clojure

Fourth Edition



Alex Miller
with Stuart Halloway
and Aaron Bedra
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 12

Project Tooling

Until now, we've been working with code largely in small units, exploring individual features of the language. However, most of your time as a developer is in the context of a *project*. Projects are not an explicit feature of Clojure but any JVM-based program has a scope implicitly defined by its *classpath*, which specifies all of the source folders and dependencies available, and effectively delineates a project scope. Projects may correspond to a single source control repository (especially for libraries), or multiple projects may be combined into a single repository.

In this chapter we'll explore the tools provided by Clojure for working with dependencies and tools in the scope of a project and everything you need to package your code for release. We won't go deep on any of the tools discussed here, we are instead demonstrating just the basics that you need for working in a project. Let's start by looking at how the JVM finds and loads your code.

Project Sources and Dependencies

A Clojure project consists broadly of two sources of code—your project code in source directories, and other people's code in external dependencies. Clojure, as a JVM language, expects both kinds of code to be on the project classpath. In this section we will explain how the JVM classpath works and how to use tools to declare the location of your source files and the dependencies you wish to use.

The JVM Classpath

The JVM classpath is a series of source file *roots*. Classpath roots can either be directories (common for your source files) or JAR (Java ARchive) files (common for external dependencies). JAR files are merely ZIP files containing their own set of files and directories.

The classpath is a string separating roots with a path separator (: on Unix, ; on Windows), for example, "src:test". When Clojure needs to load the namespace your.project.core, it looks for a source file at the path your/project/core.clj in each classpath root, in order. In this example, it will look for src/your/project/core.clj first, and then test/your/project/core.clj. The search stops when an existing file is found, and that file is loaded and compiled.

The .cljc Extension

In Clojure 1.7, the .cljc file extension was defined for *common* source files suitable for loading in multiple dialects of Clojure, with support for conditional code depending on platform. We won't cover the details in this book, but any place we talk about .clj files, .cljc files are also allowed.

Clojure is a *source-first* language. Importantly, this means that Clojure code does not need to be compiled into a binary form for distribution, making it easy to distribute Clojure as source. However, Clojure code is always compiled to JVM class files eventually, and applications may benefit from doing this ahead of time to make loading faster. We'll cover how to do this in Compiling Applications, on page 16.

When Clojure loads a namespace, it will also look to see if a compiled .class file exists in addition to checking for the .clj source file, and will load whichever is newer.

Declaring Source Roots in deps.edn

Over Clojure's history, many project management tools have come and gone—but ultimately they all provide a way to declare the project source roots, the external dependencies, so they can be combined into a classpath for the purposes of running your program. The two most common tools in use today are Leiningen[1] (a community tool) and the Clojure CLI[2] (created by the Clojure core team). In this book, we will focus on the latter.

The CLI (command line interface) uses a data file named deps.edn[3] at the project root to declare project sources and dependencies. This file is in edn[4] (Extensible Data Notation) format, which is essentially a data-oriented subset of

1. https://leiningen.org/
2. https://clojure.org/reference/clojure_cli
3. https://clojure.org/reference/deps_edn
4. https://github.com/edn-format/edn

Clojure's syntax. The deps.edn file is a map with a few well-known keys. For the purposes of declaring your source paths, this is a sufficient starting point:

```
{:paths ["src"]}
```

The :paths key is a vector of source roots to put on the classpath. Conventionally, most Clojure projects use src as the primary source root directory. If you wanted to add an additional path for resource files, you could just add a second directory path in your deps.edn:

```
{:paths ["src" "resources"]}
```

Next we will consider how to include other libraries in your project.

Maven Dependencies

Clojure has the benefit of working with either source files or class files that are either in the file system or packaged in JARs, yielding a lot of flexibility. When consuming external dependencies, the most common way to do so is by downloading a JAR of source files (for Clojure dependencies) or of class files (for Java dependencies) or possibly a mix.

For over two decades, Maven repositories have been the primary technology for Java developers to publish and consume open source and commercial libraries. From the consumption side, this is usually done via tools like mvn or gradle that take requests for a library, interact with Maven repositories, download and cache the library JAR file, then make it available to applications. We use the Clojure CLI to download and manage dependencies.

Maven artifacts are identified by coordinates made up of several parts:

- groupId—the "organization" providing the library, typically a DNS-reversed domain name or trademarked org name
- artifactId—the library name (scoped to the groupId)
- version—a string representing a version, most commonly in the form "major.minor.patch"
- classifier—an optional string representing a library variant or related artifact (not common)

The deps.edn format assembles these together into two parts:

- Library name—a symbol combining the groupId, artifactId, and classifier with the form groupId/artifactId$classifier (the classifier is optional)
- Coordinate—a map describing information about how to find the library version: {:mvn/version "..."}

Putting this all together, you can specify dependencies in the deps.edn file using the :deps key:

```
{:paths ["src"]
 :deps {
   org.clojure/clojure {:mvn/version "1.12.0"}
   org.clojure/data.json {:mvn/version "2.5.1"}
 }
}
```

This deps.edn file retains the :paths key from the prior example and adds dependencies on two libraries—Clojure itself and the data.json library.

The Clojure CLI always includes a dependency on Clojure itself via the built-in root deps.edn, so including it here is not strictly necessary, but it can be a good practice if you want to define a minimum Clojure version this project depends on.

Any time you execute the Clojure CLI, it computes the classpath from the paths and deps in the deps.edn. Computing the deps does not just involve the dependencies listed in the deps.edn, it also includes any transitive dependencies needed by the top-level dependencies, recursively. We won't go into any more detail in this book, but know that there are many ways to modify how the full classpath is computed from the top-level deps. Once the classpath has been computed, the Clojure CLI ensures every lib is downloaded and caches them in the Maven cache (usually at ~/.m2/repository).

While dependencies in the JVM ecosystem are usually released as JAR artifacts, we mentioned previously that Clojure can work directly from source, and no JAR is required. Let's look at how to use Clojure dependencies directly from a Git repository.

Git Dependencies

Git has become the de facto source control software used today, especially in open source. Providers like GitHub and GitLab have made public hosting of open source accessible to all. The majority of the libraries in the Clojure ecosystem are available in one of these open source Git hosts.

Git repositories at a specific commit or tag are effectively nothing more than a set of files. Because Clojure works from source, a public Git repository has everything we need to consume it as a Clojure library. There is no need for the author to compile source to classes, assemble class files into a JAR, publish them to a Maven repository, download the jar from the repository, and put the JAR on the classpath. Instead, a Clojure library author can

simply publish their open source project and the consumer can refer to it directly as a dependency.

In deps.edn, the library name and coordinate can be configured differently to pull in a Git dependency. The library name no longer refers to a Maven groupId and artifactId; instead the library name has these parts:

- Git host (reversed URL name)
- Organization / user
- Project

Many popular Git hosts have default mappings from a deps.edn lib name:

- GitHub: [io,com].github.ORG/PROJECT
- GitLab: [io,com].gitlab.ORG/PROJECT
- BitBucket: [io,org].bitbucket.ORG/PROJECT
- Beanstalk: [io,com].beanstalkapp.ORG/PROJECT
- Sourcehut: ht.sr.ORG/PROJECT

For the coordinate, we want a dependency to refer to a specific commit. In Git, every object is identified by a unique 40-character hexadecimal string known as the *SHA*, a hash value created with the SHA-1 algorithm (Secure Hash Algorithm 1). For convenience, Git also supports the use of short SHA prefixes if they are unique in the context of the repository. A deps.edn git dep must specify either a full commit SHA, or a short SHA and a tag name that both resolves to the same full commit SHA.

For example, the following git dep refers to the cognitect-labs.test-runner tool which we'll use later in the chapter:

```
io.github.cognitect-labs/test-runner
{:git/tag "v0.5.1" :git/sha "dfb30dd"}
```

The first line is the library name, which auto resolves to the GitHub url https://github.com/cognitect-labs/test-runner. The coordinate indicates a commit by specifying a version tag (for human understanding) and a matching short SHA for verification. Alternately a full SHA could have been used instead:

```
io.github.cognitect-labs/test-runner
{:git/sha "dfb30dd6605cb6c0efc275e1df1736f6e90d4d73"}
```

Specifying a full SHA means the commit could be anything, including a non-tagged version, or even from a branch or a fork, not just a tag on the main branch. This is a great feature for library authors because they can provide a tentative fix to a user from a development branch for testing without doing an official release. But usually, it's best to stick to a tag and version—the tag

provides semantic meaning, the version provides a check that the tag hasn't been moved (which is unusual, but possible).

Git libs are also downloaded to a git repository cache, by default in ~/.gitlibs, so the repository object cache is updated to the necessary SHA, and a checkout of the repository at the specific SHA is separately cached. In addition to the Maven local repository and Git lib cache, computed classpaths are also cached in the project .cpcache directory. In all of these cases, the data being cached is immutable and keeping it on disk allows the CLI to minimize network use and improve performance. In the common case, everything is in a cache and your program starts as quickly as possible.

Next, let's look at how to use the Clojure CLI to start a REPL, probably the mode you'll use the most.

Running a REPL

The Clojure CLI has several different execution modes for running different kinds of programs. The simplest of those is starting a REPL with the current project classpath. The CLI has two commands, clojure and clj—the latter is merely a wrapper for the former which adds command-line editing support via the rlwrap utility. Because of this, you should always use the clj to start a REPL, and no arguments are needed:

```
$ clj
Clojure 1.12.0
user=>
```

If you have not downloaded any of the dependencies before, you might also see downloads happening before the REPL starts:

```
Downloading: org/clojure/clojure/1.12.0/clojure-1.12.0.pom from central
Downloading: org/clojure/data.json/2.5.1/data.json-2.5.1.pom from central
...etc
```

These files are downloaded to the local Maven cache and subsequent executions will use them without needing to re-download the files.

Let's move on to defining and running tests.

Defining Tests

There are many good test frameworks in Clojure, but here we will only briefly cover the clojure.test library included in Clojure core. For a larger treatment of testing approaches, see *Clojure Applied [VM15]*.

By convention Clojure source code is stored in the src/ directory and unit tests are stored in the test/ directory. Because both source directories will be included on the classpath during testing, their namespaces should not overlap. Usually, this is done by taking the source namespace and adding -test to the end (which becomes _test in the file name).

For example a project with example.core namespace and example.core-test namespace will look like this:

```
src/
  example/
    core.clj
test/
  example
    core_test.clj
```

The example.core namespace might have a function in it:

```
(ns example.core)

(defn within? [min max val]
  (<= min val max))
```

And the example.core-test namespace will have some tests for the code:

```
(ns example.core-test
  (:require
    [clojure.test :refer [deftest is]]
    [example.core :as core]))

(deftest test-within?
  (doseq [x (range 0 10)]
    (is (true? (core/within? 0 10 x))))
  (is (false? (core/within? 0 10 -1)))
  (is (false? (core/within? 0 10 11))))
```

The deftest macro defines a test var test-within? and uses is to test assertions in the scope of the test.

Next we need to declare a deps.edn file that adds the test/ directory to the project paths (we'll see a better way to do this soon, but this will work for now):

```
{:paths ["src" "test"]}
```

And then you can run these tests by starting the REPL with clj, loading the test namespace, and calling run-tests:

```
(require 'example.core-test 'clojure.test)
-> nil
(clojure.test/run-tests 'example.core-test)
|
| Testing example.core-test
```

```
|
| Ran 1 tests containing 12 assertions.
| 0 failures, 0 errors.
-> {:test 1, :pass 12, :fail 0, :error 0, :type :summary}
```

Once you start to accumulate more than one test namespace, this quickly gets cumbersome. Most Clojure developers rely on helper tooling in their editor environments to run all of the tests in a namespace or project, and project tools to run all tests from the command-line or in continuous integration.

Next we'll look at how to set up a command-line test runner in deps.edn using the test-runner tool.

Running Tests

The Clojure CLI does not come with a built-in test runner and instead provides generic support for build tools, including test runners. One of the most commonly used clojure.test test runners is cognitect-labs.test-runner.[5]

To set up the test runner, we need to introduce another feature of deps.edn, *aliases*. Aliases associate a name (the alias) with an arbitrary map of edn data. The Clojure CLI defines a special set of alias keys that can modify the behavior of the CLI itself.

We declare the alias in the deps.edn like this:

```
{:paths ["src"]
 :aliases {
   :test {:extra-paths ["test"]
          :extra-deps {io.github.cognitect-labs/test-runner
                       {:git/tag "v0.5.1" :git/sha "dfb30dd"}}
          :exec-fn cognitect.test-runner.api/test}
 }
}
```

Note that we're not including the "test" directory in :paths—instead, everything we use for testing will be isolated in the :test alias. The new keys here are:

- :extra-paths—paths to add to :paths when this alias is used
- :extra-deps—deps to add to :deps when this alias is used, here as a git dep (check the project to find the newest version available)
- :exec-fn—the default tool function to invoke (some tools have multiple functions)

5. https://github.com/cognitect-labs/test-runner

To run the test-runner we also need to introduce a new "mode" of the Clojure CLI, *function execution mode*, triggered with -X. The alias (or aliases) to use are appended to the -X option like this:

```
$ clj -X:test

Running tests in #{"test"}

Testing example.core-test

Ran 1 tests containing 12 assertions.
0 failures, 0 errors.
```

Passing -X executes a function, usually in a tool, but potentially also in the project itself. Appending :test specifies the alias data to activate in deps.edn, which modifies the classpath by adding the test source path and test-runner dependency. The :exec-fn specifies the test-runner function to run. The default behavior of test-runner is to look for namespaces ending in -test and run all of the tests it finds. The example above finds and runs the only test namespace example.core-test.

The test-runner tool has other options to run specific namespaces, subsets of marked namespaces, specific vars, and more but we won't cover those here.

Now that you know how to set up your project paths and deps, start a REPL, and run tests, let's look at how to build and release a library for others to use.

Building and Releasing Libraries

Clojure libraries can be released to either Maven or Git (or both), and there are some tradeoffs to consider. We'll discuss Git deployment first, which is easier but has some usage limitations, and then cover Maven deployment which is more complicated but usable by more Clojure and Java tools.

Git Deployment

If you are developing an open source Clojure library, you are likely already managing it in a Git host like GitHub. If so, you are almost done deploying it! Because no compilation or artifact assembly is required, this mostly requires a few steps which you may already be doing:

- Create and publish a public Git repo in your account
- Include a deps.edn file at the project root (there is also support for nested projects but we'll assume here the project root is also the repository root)
- Tag the repo to create a version—while this is optional, it's a good practice

You can create a tag for your project and find its short SHA with git commands like:

```
# Create an annotated tag with a message
# Using "vX.Y.Z" is a common practice for version tags

git tag -a 'v0.1.0' -m 'v0.1.0'

# Push the tags to the upstream repository

git push tags

# Obtain the short SHA version

git rev-parse --short v0.1.0^{commit}
```

The last line uses `^{commit}` to unpeel the tag to its related commit SHA, which should match what you see in the Git host user interface.

You now have all the information you need to publish your Git lib and coordinate for users to use—the Git library name is based on the Git host, user, and project and the coordinate is the tag and short SHA.

With a bit of additional work, you can easily automate this into something like a GitHub action and make tagging quick and easy.

The major limitation with Git libraries is that this feature is only available in the Clojure CLI so all downstream users must also be using the Clojure CLI. As mentioned previously, Clojure developers may use other tools like Leiningen, and publishing only a Git library limits their access to the library. Also, if you want other language communities to make use of the library, tools like Maven and Gradle in Java or sbt in Scala won't understand how to use this library.

For these reasons, most Clojure developers use Git libraries primarily when publishing Clojure CLI tools (like the `test-runner` we saw earlier) or when producing libraries mostly for their own use.

To be maximally available, library authors can create a JAR and publish to a Maven repository. All JVM-based dependency management tools understand how to download and use JARs in Maven repositories.

Maven Repositories

The Maven ecosystem has been around for over two decades and is its own sprawling and complex system. Maven repositories can be created and supported in many ways and there are dozens available for niche parts of the library ecosystem. However, as Clojure developers, there are really only two that are important—Maven Central and Clojars.

Maven Central is, as its name implies, a "central" repository and the default repository used by every Maven-based tool. Maven Central was started by the same authors as Maven itself and is the oldest and largest Maven repository, with a huge number of libraries, both open-source and commercial. Clojure itself and all of the Clojure contrib libraries are published on Maven central. Maven Central requires uploading a variety of auxiliary artifacts, signing all artifacts using public-private key cryptography, and an account authenticating that you own or control the groupId you plan to use. In other words, deployment is fairly complicated.

Clojars is a Clojure-centric Maven repository started in the early days of Clojure and still run by the community. The rules for artifact and groupId construction are somewhat looser and the process is in general much easier to manage. For this book, we are going to focus on deploying to Clojars, but deploying to Maven Central is similar—it just requires a few extra steps.

Artifact Building

We will produce our project artifact with tools.build, a library for writing build programs. Once we can build the artifact, we will deploy it to Clojars. Clojars will only allow the deployment of artifacts under names we control (which affects how we build the artifact in the first part), so let's take a brief detour to create an account on Clojars.

Go to https://clojars.org/ and click Register. Create an account and password to use as your Clojars login. You will use your Clojars user name in the library name below.

Next, let's look at how to create a build.clj build program using tools.build. The main steps in building a library artifact are:

- Load the project basis (data representing the project classpath)
- Copy the Clojure source and resources into a working directory
- Write a Maven pom file to the working directory (this file tells Maven what this artifact depends on)
- Zip the working directory into a JAR file

The full build.clj script will look something like this:

```
(ns build
  (:require [clojure.tools.build.api :as b]))

(def class-dir "target/classes")
(def lib 'org.clojars.USER/LIB)  ;; Insert your own Clojars user and lib name
(def license-data
  [[:licenses
```

```
    [:license
      [:name "Eclipse Public License 1.0"]
      [:url "https://opensource.org/license/epl-1-0/"]]]])

(defn jar [{:keys [version]}]
  (b/delete {:path "target"})
  (b/write-pom {:class-dir class-dir, :src-dirs ["src"],
                :lib lib, :version version,
                :basis (b/create-basis {:project "deps.edn"})
                :pom-data license-data})
  (b/copy-dir {:src-dirs ["src"], :target-dir class-dir})
  (b/jar {:class-dir class-dir, :jar-file "target/lib.jar"}))
```

You should also include the :scm properties linking your artifact to your code if it is open source. For brevity we have not included these here. Refer to the docs[6] for these and other optional Maven properties.

The build.clj script is placed at the root of the project, and then deps.edn needs a new alias to run this build script with tools.build:

```
{:paths ["src"]
 :aliases {
   :test {:extra-paths ["test"]
          :extra-deps {io.github.cognitect-labs/test-runner
                       {:git/tag "v0.5.1" :git/sha "dfb30dd"}}
          :exec-fn cognitect.test-runner.api/test}
   :build {:deps {io.github.clojure/tools.build {:mvn/version "0.10.9"}}
           :ns-default build}
 }
}
```

Here we need to introduce a new Clojure CLI execution mode for running *tools*. Tools are useful programs for your project that have their own classpath and don't include the project classpath. In this case the program we are running is the build.clj itself, which uses tools.build as a library.

To execute the build as a tool we use -T instead of -X:

```
clj -T:build jar :version '"0.1.0"'
```

Note that we specify both the alias (:build) and the function to run in the build.clj script (jar). The build script might contain other build functions as well. Also, we need to pass the version to build as an argument, here version "0.1.0".

When the build completes, the JAR archive will exist at target/lib.jar. This file contains the manifest files defining a Maven JAR file and the Clojure source files from the project.

6. https://clojure.github.io/tools.build/clojure.tools.build.api.html#var-write-pom

Once we've created the archive, we need to deploy it to Clojars.

Clojars Deployment

It's now time to add one more alias to our deps.edn to deploy the built jar using another tool, slipset/deps-deploy. Here is the deps.edn with that added alias :deploy:

```
{:paths ["src"]
 :aliases {
   :test {:extra-paths ["test"]
          :extra-deps {io.github.cognitect-labs/test-runner
                       {:git/tag "v0.5.1" :git/sha "dfb30dd"}}
          :exec-fn cognitect.test-runner.api/test}
   :build {:deps {io.github.clojure/tools.build {:mvn/version "0.10.9"}}
           :ns-default build}
   :deploy {:extra-deps {slipset/deps-deploy {:mvn/version "0.2.2"}}
            :exec-fn deps-deploy.deps-deploy/deploy
            :exec-args
              {:installer :remote
               :sign-releases? false  ;; skip GPG signing for now
               :artifact "target/lib.jar"
               :pom-file
               "target/classes/META-INF/maven/org/clojars/USER/LIB/pom.xml"}}
 }
}
```

The alias refers to the jar and pom file that were created by the build—these contain sufficient info about the library and version to deploy to the Clojars repository. Note that the :pom-file directory will depend on your Clojars user and lib name.

If you have not yet created a deploy token in Clojars, log into the site and go to "Deploy Tokens" in the menu to create a user token—make sure to save this as it will only be shown once.

Now we're ready to deploy the artifact to Clojars. Export environment variables for the user and password (use the token), then invoke the :deploy tool:

```
$ export CLOJARS_USERNAME=username
$ export CLOJARS_PASSWORD=clojars-token
$ clj -X:deploy
```

If everything goes smoothly, you should quickly see your library appear in the Clojars site. At that point, other users can consume the jar using the Maven coordinates from Clojars.

In general, Clojure libraries are usually distributed as source. Compiling Clojure library code makes some early decisions that are generally better left to the application, namely which Clojure version and JDK version to compile

against. The Clojure version commits to a specific version of the compiler, which may not match the Clojure version used by an application. Clojure compiled code is generally forward-compatible but it's not necessarily backwards-compatible so you have narrowed the applicability of your library. Similarly, the JDK version available at compile time affects Java interop choices and may make different method resolution choices than would be made on a different JDK version.

We've now completed the final step for a library: deployment of the source jar to the Clojars Maven repository. Next we will look at a place where compilation does make sense—in an application.

Compiling Applications

While compilation of Clojure libraries is unusual, compilation of Clojure applications is often a great choice that moves compile costs to build time so that the application can start more quickly at runtime. There are no issues with Clojure or JDK version, you've already made those choices at the application level and won't need to use them in other contexts.

To compile an application, we will imagine a new project with a different build.clj. In this modified build script, we need to introduce a new build task to compile the Clojure source. Clojure compilation is transitive and will also compile any code transitively loaded by your application, including library code. One of the critical choices is specifying which namespaces should be compiled, and these should include any main entry points to the application.

First, make sure that your application main entry point contains a (:gen-class) clause and a -main function so that it will be compiled to a Java class with a main suitable for direct invocation:

```
(ns example.main
  (:gen-class))

;; This must be named -main
(defn -main [& args]
  (do-stuff))
```

Here's what the modified build.clj will look like:

```
(ns build
  (:require [clojure.tools.build.api :as b]))

(def class-dir "target/classes")
(def basis (b/create-basis {:project "deps.edn"}))

(defn app [_]
  (b/delete {:path "target"})
```

```
(b/copy-dir {:src-dirs ["src"], :target-dir class-dir})
(b/compile-clj {:basis basis, :ns-compile '[example.main],
                :class-dir class-dir})
(b/uber {:class-dir class-dir, :uber-file "target/app.jar"
         :basis basis, :main 'example.main}))
```

Notice that some of the things we did for the library build are no longer needed. We don't need a pom file if we're not deploying to Maven and we don't need to build a library jar file. Instead, we add two new tasks: compile-clj to compile the example.main namespace into the class-dir, and the uber task to build a single jar containing all of the dependencies plus the compiled source into a single target/app.jar.

We will run it just like we ran the build before:

```
clj -T:build app
```

When this completes, the self-contained application will be in target/app.jar and you can run it like this:

```
java -jar target/app.jar
```

Once you've constructed this "uber" jar, you can run it on your own machine, or on a physical or virtual host. You will only need Java and your application uber jar. It's even possible to use Java jpackage utility to create a standalone application or the GraalVM JDK to create a native application (particularly good for fast starting applications). The tradeoff between these approaches is generally extra time and effort for building the application artifact vs improved size and startup time of the application. Most Clojure users run directly from an uber jar, so we will not cover the jpackage or GraalVM approaches further here.

Wrapping Up

Whew! That was a whirlwind tour through basic project tooling, including an overview of how to declare source paths and dependencies, run the REPL, write and run tests, build and deploy libraries to Clojars, and compile and build application uber jars.

Next, we will build an application and put everything we've learned so far to use!