

**Malleus Formicarum**  
*or, The Ants Hammer*  
**Leiningen Users Guide**



---

# Notice

---

## Notice

---

### Topics:

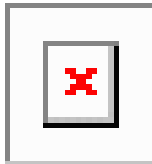
- [Trademarks](#)

This information was developed for products and services offered in the U.S.A.

This product is meant for educational purposes only. Some of the trademarks mentioned in this product appear for identification purposes only. Not responsible for direct, indirect, incidental or consequential damages resulting from any defect, error or failure to perform. Any resemblance to real persons, living or dead is purely coincidental. Void where prohibited. Some assembly required. Batteries not included. Use only as directed. Do not use while operating a motor vehicle or heavy equipment. Do not fold, spindle or mutilate. Do not stamp. No user-serviceable parts inside. Subject to change without notice. Drop in any mailbox. No postage necessary if mailed in the United States. Postage will be paid by addressee. Post office will not deliver without postage. Some equipment shown is optional. Objects in mirror may be closer than they appear. Not recommended for children. Your mileage may vary.

No other warranty expressed or implied. This supersedes all previous notices.

### **COPYRIGHT LICENSE:**



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

## Trademarks

---

The following terms are trademarks of the Foobar Corp. in the United States, other countries, or both:

Foobarista<sup>®</sup>

The following terms are trademarks of other companies:

Red, Orange, Yellow, Green, Blue, Indigo, and Violet are registered trademarks of Rainbow Corporation and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

# Contents

Abstract.....	9
Preface: About Leiningen.....	xi
Plan of the Work.....	xi
Chapter 1: Introduction: Configuration, Customization, Coordination.....	13
The Problem.....	14
The Solution.....	14
Chapter 2: What is Leiningen?.....	15
Structure & Dynamics of Leiningen Projects.....	16
Project Maps.....	16
Commands.....	16
The Leiningen Infrastructure.....	17
Getting Started.....	17
Installation.....	17
A Quick Tour.....	17
Part I: Using Leiningen.....	23
Chapter 1: Parameters and Options.....	25
Chapter 2: Structure and Interpretation of Leiningen Projects.....	27
Chapter 3: The Project Map.....	29
Profiles.....	30
System Profiles.....	30
User-defined Profiles.....	30
Chapter 4: Commands.....	31
Preinstalled tasks.....	32
User-defined Tasks: Plugins.....	33

Chapter 5: Dependency Management.....	35
Dependency Trees.....	36
Namespaces.....	36
Classpath.....	36
Chapter 6: Repositories.....	37
Chapter 7: Templates.....	39
Chapter 8: REPL Workflow.....	41
Chapter 9: Trampolines.....	43
Chapter 10: Testing.....	45
Unit Testing.....	46
Unit Testing Concepts.....	46
Unit Testing Procedure.....	46
System Testing.....	47
System Testing Concepts.....	47
System Testing Procedure.....	47
Continuous Integration.....	47
Continuous Integration Concepts.....	48
Continuous Integration.....	48
Chapter 11: Deployment.....	49
Part II: Extending Leiningen.....	51
Chapter 1: Template Development.....	53
Chapter 2: Plugin Development.....	55
Chapter 3: Profile Development.....	57
Chapter 4: Embedding Leiningen.....	59

Appendices.....61

Appendix A: Contributing to this document.....63





# Abstract

---

User Guide for Leiningen. See also the Technical Manual.



---

# Preface

---

## About Leiningen

---

### Plan of the Work

---

The basic approach: describe the problem, then describe general strategies for addressing the problem, then describe Leiningen-specific solutions, then give an example.

Key distinctions: Leiningen “task” (i.e. command) v. developer's task (e.g. test, document); project map (i.e. configuration) v. command

👉 Note: Leiningen's terminology should be changed to use “command” where it currently uses “task”. This is because the tasks are not in fact tasks; e.g. `lein compile` is a command from the user to Leiningen rather than a task to be accomplished. We need to reserve the word “task” to refer to “user tasks”, the tasks confronting the developer, which appear as problems to be solved. So for example the first task is merely to configure the project. The way to solve this problem is to write the `project.clj` and set up the directory structure.

Plan of the work:

Overview	installation, projects, infrastructure
Project Maps	How the projmap is constructed at runtime; how tasks use the projmap to drive configuration of the
Tasks	Two kinds of task: <ul style="list-style-type: none"><li>• Dev tasks, e.g coding, testing, QA, etc</li><li>• Leiningen “tasks”, i.e. commands</li></ul> General description of tasks; commonly used tasks with examples
Infrastructure: Respositories	role of repos; how to specify, etc. and how Leiningen searches, etc
etc	etc



---

# Chapter 1

---

## Introduction: Configuration, Customization, Coordination

---

### Topics:

- [The Problem](#)
- [The Solution](#)

"Leiningen is for automating Clojure projects without setting your hair on fire."

Leiningen is a tool for:

**Configuration** Use it to "configure" tools, i.e. set options, for various tasks.

**Customization** Groups and individuals can "inherit" general configurations and then customize them on a per-task and/or per-project basis.

**Coordination** Use it to integrate and coordinate related activities. For example, quality assurance, code check-in, and issue tracking are related activities. Leiningen can help you automate standard procedures such as: first QA check code, then check it in with a reference to the bug it fixes, and then close the bug in the Issues Tracker, annotating it with the commit ID.

## The Problem

---

Leiningen is a general “CCC” (configuration, customization, and coordination) tool.

This section describes the sort of problems Leiningen is designed to solve: configuration, coordination, etc.

See this thread <https://groups.google.com/forum/#!topic/clojure/OiAijeVr6k>

## The Solution

---

This section describes in general how Leiningen solves the problems described in the previous topic. I.e. it's more about the strategies Leiningen adopts than the specific implementation techniques; they are the subject of the remainder of this book..

etc

---

# Chapter 2

---

## What is Leiningen?

---

### Topics:

- [Structure & Dynamics of Leiningen Projects](#)
- [Getting Started](#)

Leiningen is a collection of commands that address developer tasks and needs. It depends on a kernel of core functionality, a set of plugins that implement the commands available to users, and a set of external resources - directories, files, maven repositories, etc.

Leiningen can be compared to Emacs. Emacs consists of a kernel of core functionality (implemented in C) made available to the user by means of commands or functions defined in Elisp, just like user-defined functions and commands. Leiningen also has a kernel of core functionality (implemented in Clojure) that is made available to users by means of *plugins*, which define Leiningen *commands*; user-defined commands are implemented in the same way.

## Structure & Dynamics of Leiningen Projects

---

Structure of project: project config map (in `project.clj`) determines dir structure.

Dynamics: when a lein command is invoked, lein dynamically constructs the effective project map and invokes the command, passing the EPM as arg. The command inspects the map and uses its content to control its processing.

### Project Maps

The central concept of Leiningen is the *project map*.

What does a project map look like? An ordinary Clojure map: set of key-val pairs. In the project map, keys are always Clojure `:-`-prefixed keys. L defines a default set of keys; users are free to extend this set.

#### The Project Configuration Map

The *project configuration map* is defined by `defproject` macro in the `project.clj` file in the project *root directory*.

Terminology: the *project configuration map* is specified as part of `defproject` in `project.clj`. The *effective project map* is dynamically constructed by combining the project config map and various other maps - see X for details.

#### Profiles

Profiles provide a means of customizing the effective project map.

A profile is a named map. Leiningen predefines several *system profiles*: `:base`, `:system`, `:user`, `:dev`, and `:provided`. The system profiles are always in effect; user-defined can override and/or extend them. User-defined profiles can be specified in several places:

At runtime, Leiningen integrates profiles with the project configuration map of `project.clj` to produce the effective project map.

#### The Effective Project Map

The *effective project map* is dynamically constructed by combining the project config map and various other maps

Leiningen command implementations receive the effective project map as an argument when they are invoked.

General description of how L constructs the *effective project map* from `project.clj/defproject` in combination with various other maps. This is one of the basic jobs of the kernel. For details, refer to the Project Map node.

#### Project Map Semantics

Project map semantics are determined by command implementations.

What are the semantics of the (effective) project map? Determined entirely by the command implementations. All Leiningen commands use the effective project map, which is delivered by Leiningen to the command as its sole argument. Different commands pick out different parameters from the map for use in controlling processes.

### Commands

Leiningen *commands* (formerly “tasks”) expose functionality to the user.

Commands are what you would expect: something you type to tell lein what to do.

L comes with a set of predefined commands. Users can extend this set by writing a *plugin*, which is the implementation of a command. There is no technical difference between L's predefined commands and user-defined commands.



## The Leiningen Infrastructure

Leiningen's infrastructure config files, the local repo, remote repos, etc. We should also include default profiles such as `:base`, since they are just as fundamental. This topic provides a brief overview of these parts and how they fit together.

### External

Directories, files, env vars used by Leiningen

Directories: `~/.lein`, `~/lein/profiles.d`, etc; `~/.m2/repository`;

Files: `~/.lein/profiles.clj`

Environment variables

Command line options

Anything else Leiningen can take from the env?

Do this

### Internal infrastructure: profiles, etc.

Profiles and other internally defined (but overridable) stuff on which Leiningen's functionality depends.

## Getting Started

---

We get started by first exploring a minimal project, and then exploring the Leiningen infrastructure. This should give us a good general idea of what pieces are involved and how the work together.

Before delving into the details we take a brief tour of:

- A minimal project
- The Leiningen infrastructure

## Installation

### Unix-like systems

If your preferred [package manager](#) has a relatively recent version of Leiningen, try that first. Otherwise you can install by hand:

Leiningen bootstraps itself using the `lein` shell script; there is no separate install script. It handles installing its own dependencies, which means the first run will take longer.

1. Make sure you have a Java JDK version 6 or later.
2. [Download the script](#)
3. Place it on your `$PATH`. (`~/bin` is a good choice if it is on your path.)
4. Set it to be executable. (`chmod 755 ~/bin/lein`)

### Windows

There is an [installer](#) which will handle downloading and placing Leiningen and its dependencies.

The manual method of putting the [batch file](#) on your `PATH` and running `lein self-install` should still work for most users. If you have Cygwin you should be able to use the shell script above rather than the batch file.

## A Quick Tour

We create a minimal project and explore its structure and configuration.

A minimal Leiningen project involves: a directory structure, a project configuration file (`project.clj`), and source code files. In addition to these static resources, Leiningen dynamically determines a project map, a dependency tree, and several other structures. This section takes you through some simple steps to explore these elements.

1. Create a new project by executing: `lein new app my-app`

where `lein` is the Leiningen command, and the args are:

`new`

a Leiningen *task*. To see the syntax and semantics of this task, run `lein help new`. To see a list of the tasks that come preinstalled, run `lein help`.

`app`

is the name of a project template, in this case the default application template. To see a list of preinstalled templates, run `lein help new`. Leiningen's concept of *task* is discussed in detail under the X topic of this guide.

`my-app`

is the name to be used for the new project

This will create a hierarchy of directories and populate it with some files generated from templates:

```
.
./ .gitignore
./ doc
./ doc/intro.md
./ project.clj
./ README.md
./ src
./ src/my_app
./ src/my_app/core.clj
./ test
./ test/my_app
./ test/my_app/core_test.clj
```

2. Take a look at the Leiningen project file by running `less my-app/project.clj`  
You should see something like:

```
(defproject my-app "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

👉 **Note:** This has the form of a function application, where `defproject` is the function, and the args come as a list of pairs. In fact, `defproject` is a macro that expands into the definition of a Clojure map named `project`. Each task - both those preinstalled and those defined by plugins - takes this `project` map as input. The key point is that the key-value pairs of the `project` map are thus available for use by task implementations to control setting of options etc. for the processes they manage.

👉 **Note:** The final `project` map is determined by a combination of several sources in addition to `project.clj`, such as `~/.lein/profiles.clj`. See X for details.

In this minimal example, the only really functional parameter is `:dependencies`. This is used to specify which libraries/jars the project uses (and thus depends on). One of the best things about Leiningen is its powerful management of dependencies. Using the `[?]` library under the

hood, Leiningen is able to construct the entire dependency graph for the project and arrange for everything needed to be installed. Leiningen's dependency management capabilities are described in X.

👉 Important: Leiningen dependency specification uses the naming conventions established by Maven: artifact-group/artifact-id. Etc. See X for details.

### 3. Examine the project map: `lein ppprint`

You should see something like:

```
{:compile-path "/Users/gar/tmp/my-app/target/classes",
:group "my-app",
:license
{:name "Eclipse Public License",
:url "http://www.eclipse.org/legal/epl-v10.html"},
:global-vars {},
:checkout-deps-shares
[:source-paths
:test-paths
:resource-paths
:compile-path
#&Var@3e25e2b8:
#&classpath$checkout_deps_paths
leininingen.core.classpath$checkout_deps_paths@89bbe8c>>],
:dependencies
([org.clojure/clojure "1.5.1"]
[org.clojure/tools.nrepl "0.2.3":exclusions ([org.clojure/clojure])]
[clojure-complete/clojure-complete
"0.2.3"
:exclusions
([org.clojure/clojure]))),
:plugin-repositories
[["central" {:snapshots false, :url "http://repo1.maven.org/maven2/"}]
["clojars" {:url "https://clojars.org/repo/" }]],
:test-selectors {:default (constantly true)},
:target-path "/Users/gar/tmp/my-app/target",
:name "my-app",
:deploy-repositories
[["clojars"
{:username :gpg,
:url "https://clojars.org/repo/",
:password :gpg}]],
:root "/Users/gar/tmp/my-app",
:offline? false,
:source-paths ("/Users/gar/tmp/my-app/src"),
:certificates ["clojars.pem"],
:version "0.1.0-SNAPSHOT",
:jar-exclusions [#"^\\.\\.\\."],
:profiles {:uberjar {:aot :all}},
:prep-tasks ["javac" "compile"],
:url "http://example.com/FIXME",
:repositories
[["central" {:snapshots false, :url "http://repo1.maven.org/maven2/"}]
["clojars" {:url "https://clojars.org/repo/" }]],
:resource-paths
("/Users/gar/tmp/my-app/dev-resources"
"/Users/gar/tmp/my-app/resources"),
:uberjar-exclusions [#"(^|META-INF|\\.\\.\\.)(SF|RSA|DSA)$"],
:main my-app.core,
:jvm-opts ["-XX:+TieredCompilation" "-XX:TieredStopAtLevel=1"],
:eval-in :subprocess,
:plugins
```

```
([lein-localrepo/lein-localrepo "0.4.1"]
 [lein-diffest/lein-diffest "1.3.8"]
 [org.clojure/java.classpath "0.2.0"]
 [lein-marginalia/lein-marginalia "0.7.1"]
 [lein-mustache/lein-mustache "0.2"]
 [lein-pprint/lein-pprint "1.1.1"]),
:native-path "/Users/gar/tmp/my-app/target/native",
:description "FIXME: write description",
:test-paths ("/Users/gar/tmp/my-app/test"),
:clean-targets [[:target-path]],
:aliases nil}
```

Take some time to look this over. Most of these parameters you will never have to deal with, but it's good to have an idea of what sorts of things Leiningen is interested in.

👉 Note: The predefined parameters are documented in the Technical Reference Manual. Other chapters of this User's Guide explain how to use them.

👉 Note: We should distinguish between the project map that results from Leiningen's work and the `defproject` map in `project.clj` that forms the starting point for project map construction.

#### 4. Tell Leiningen to install all dependencies by running `lein deps`

Since you already have Clojure installed, you probably won't see any output.

#### 5. Check your dependency tree: `lein deps :tree`

You should see something like the following:

```
[clojure-complete "0.2.3" :exclusions [[org.clojure/clojure]]]
[org.clojure/clojure "1.5.1"]
[org.clojure/tools.nrepl "0.2.3" :exclusions [[org.clojure/clojure]]]
```

This is a complete listing of the jars your project depends on, derived from your `project.clj` `:dependencies` parameter plus a set of default maps to be described later. Since this is a tree representation, you can infer that these three libraries are independently specified; none of them has required any of the others as a dependency. In fact, `clojure-complete` and `org.clojure/tools.nrepl` are both specified as dependencies by the default `:base profile`, which means that they will always be in the dependency tree for every project (unless overridden). Profiles are named maps that can be used to customize the `project` map in various ways; they are fully described in X.

#### 6. Now let's take a look at the application code installed by the `app` template. Use your editor, or run `cat src/my_app/core.clj`

The contents of `my-app/src/my_app/core.clj` should look something like:

```
(ns my-app.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

👉 Notice: The `:gen-class` option and the definition of `-main`. Remember this was produced by the application template, rather than the library template. So it assumes you want to execute the result, which means you need to generate Java byte code. That's what `:gen-class` does. You also need a main entry point; that's what `-main` does.



Warning: In case it isn't obvious: to effectively use Leiningen, you need to know Clojure. That is, the better your mastery of Clojure, the more you can do with Leiningen. Leiningen is a Clojure application, after all.



---

# Part

## I

---

## Using Leiningen

---

### Topics:

- *Parameters and Options*
- *Structure and Interpretation of Leiningen Projects*
- *The Project Map*
- *Commands*
- *Dependency Management*
- *Repositories*
- *Templates*
- *REPL Workflow*
- *Trampolines*
- *Testing*
- *Deployment*





---

# Chapter 1

---

## Parameters and Options

---

Terminology: in keeping with standard Unix usage, we reserve the term *option* for bits of state that control the behavior of software. For example, many command line tools have a “verbose” option that controls how much information the tool dumps the stdout/stderr as it executes.

In order to specify (or set) an option, the user must supply the appropriate *parameter* as part of the command. Alternatively, software commonly accepts parameters stored in environment variables. Yet a third alternative is to accept parameters specified in a configuration file.

In practice the distinction between options and parameters in this sense is overlooked or not even noticed in the first place, with no ill effect. But for the sake of clarity, it is useful to respect the distinction when it comes to Leiningen. There are two reasons for this.

First, Leiningen project files may contain parameters for any number of programs, each of which support different options. So although it makes sense to think of the project map as a kind of configuration file, it is not like the configuration files specific to particular software packages. For example, Jetty depends heavily on configuration files that are specific to Jetty; there is no reason to put anything in a Jetty configuration file that Jetty would not understand. A Leiningen project map, by contrast, ordinarily will contain parameters for several different software components.

Second, Leiningen places no constraints on the representation of any option. Since Leiningen is extensible via plugins, the most Leiningen can do is provide a means of associating a value with a key - a Clojure map. It then becomes the responsibility of the plugin to correctly interpret the key-value pair (which we call a *parameter*). This means in particular that a plugin designer can choose key and value names for configuration parameters that are completely different than their corresponding software options.

A simple example: the `java` command understands a “max heap size” option, whose command line parameter is `-Xmx<size>`. Leiningen does not directly support this as a configuration parameter, however; instead it has a `:jvm-opts` parameter, whose value can be anything `java` understands to be an option-setting parameter. So Leiningen must *construct* the appropriate command line from the project map parameters.

Furthermore, it is possible to choose completely different names; for example, Leiningen could have used “semantic” parameters; in this example, it could have supported a `:jvm-max-heap` parameter that takes a number in MB as a value; or, it could have used a nested map, e.g. `:jvm-opts {:heap {:max 512 :min 64}}`.

So for the sake of clarity we will always refer to the key-value pairs in the project map as *parameters*, and refer to the software state controlled by such parameter A given parameter is not necessarily specific to a particular software component, whereas options are always software-specific.

---

# Chapter 2

---

## Structure and Interpretation of Leiningen Projects

---

Abstract: a project in Leiningen has a static structure and a dynamic (runtime) interpretation.

Structure: the project config map + profiles

Interpretation: combination of project config map and profiles



---

# Chapter

# 3

---

## The Project Map

---

### Topics:

- [Profiles](#)

The central concept of Leiningen is the *project map*.

What does a project map look like? An ordinary Clojure map: set of key-val pairs. In the project map, keys are always Clojure `:-`-prefixed keys. L defines a default set of keys; users are free to extend this set.

How does the project map function?

The effective project map computed at runtime - how?

## Profiles

---

[Overview ...](#)

### System Profiles

Several profiles are predefined by Leiningen:

#### **:base**

The `:base` profile

#### **:system**

The `:system` profile

#### **:user**

The `:user` profile

### User-defined Profiles

Users (and plugin developers) can define their own *user-defined profiles*. User-defined profiles take effect *after* system profiles; they override and/or extend the system profiles.

---

# Chapter 4

---

## Commands

---

### Topics:

- [Preinstalled tasks](#)
- [User-defined Tasks: Plugins](#)

Leiningen *commands* (formerly “tasks”) expose functionality to the user.

Commands are what you would expect: something you type to tell lein what to do.

L comes with a set of predefined commands. Users can extend this set by writing a *plugin*, which is the implementation of a command. There is no technical difference between L's predefined commands and user-defined commands.

## Preinstalled tasks

---

Overview ...

Leiningen 2.3.2 Standard  
Tasks

check	Check syntax and warn on reflection.
classpath	Write the classpath of the current project to output-file.
clean	Remove all files from paths in project's clean-targets.
compile	Compile Clojure source into .class files.
deploy	Deploy jar and pom to remote repository.
deps	Show details about dependencies.
difftest	
do	Higher-order task to perform other tasks in succession.
help	Display a list of tasks or help for a given task or subtask.
install	Install current project to the local repository.
jar	Package up all the project's files into a jar file.
javac	Compile Java source files.
localrepo	Work with local Maven repository
marg	Run Marginalia against your project source files.
mustache	Evaluate a Mustache template.
new	Generate scaffolding for a new project based on a template.
plugin	DEPRECATED. Please use the :user profile instead.
pom	Write a pom.xml file to disk for Maven interoperability.
pprint	Pretty-print a representation of the project map.
repl	Start a repl session either with the current project or standalone.
retest	Run only the test namespaces which failed last time around.
run	Run the project's -main function.
search	Search remote maven repositories for matching jars.
show-profiles	List all available profiles or display one if given an argument.
swank	obsolete??
test	Run the project's tests.
trampoline	Run a task without nesting the project's JVM inside Leiningen's.
uberjar	Package up the project files and all dependencies into a jar file.
update-in	Perform arbitrary transformations on your project map.
upgrade	Upgrade Leiningen to specified version or latest stable.
version	Print version for Leiningen and the current JVM.
with-profile	Apply the given task with the profile(s) specified.



## User-defined Tasks: Plugins

---

Overview ...



---

# Chapter 5

---

## Dependency Management

---

### Topics:

- [Dependency Trees](#)
- [Namespaces](#)
- [Classpath](#)

Compare: in C (makefiles), dependencies are source files. In Leiningen, dependencies are jars.

Problem: how to identify dependencies? Solution: use Leiningen's convention of "GROUP-ID/ARTIFACT-ID "VERSION"" is based on Maven. See [Maven Glossary](#), [Maven Guide to naming conventions on groupId, artifactId and version](#).

Use of `lein deps` and `lein deps :tree`.

Problem: how does Leiningen search? Soln: describe `:repos`. Something about using local repos.

Problem: what to do if Leiningen fails to find the dependency. Interpreting error messages.

### Dependency Versions

The problem: how do I automatically make sure I'm using the latest versions of my dependencies?

See [this stackoverflow question](#) about using the latest version of deps.

## Dependency Trees

---

How does Leiningen figure out the dependency tree? How does it manage to fetch all the required jars? How does it manage to add stuff to the classpath?

- Pomegranate* Clojure wrapper for Aether. Aether is what handles dependency resolution (i.e. figuring out the transitive dependency graph).
- Aether* “Aether is a library for working with artifact repositories. Aether deals with the specification of local repository, remote repository, developer workspaces, artifact transports, and artifact resolution.” For details on how Aether determines the dependency graph see [Aether/Transitive Dependency Resolution](#) and [Aether/Dependency Graph](#).
- Classlojure* “classlojure lets you easily create a classloader with an alternate classpath and evaluate clojure forms in it. This classloader can even use a different version of clojure than your primary classloader.”
- Bultitude* “Bultitude is a library for finding namespaces on the classpath.”



Note: How much of all this does the user really need to know?

## Namespaces

---

### Classpath

---

Lein 1 put all deps in lib, which is on the classpath. Lein 2 puts them all in ~/.m2/repository. At run time, Leiningen arranges to add the appropriate filepaths to the jars in ~/.m2/repository to the classpath.

This can be problematic in some circumstances. For example, Google App Engine uses a modified version of Jetty that disallows loading code from outside of the `war/WEB-INF` directory, so references to jars under ~/.m2/repository have no effect. In this case you must copy the jars to `war/WEB-INF/lib`.

```
lein classpath
```

---

# Chapter 6

---

## Repositories

---

Local, remote; Clojars



---

# Chapter 7

---

## Templates

---

All you really need to know is `lein new`.

A word here about how Leiningen finds templates, and about clojars, so the user can troubleshoot in case something goes wrong.

See the Developer's Manual for information on how to write your own templates.





---

# Chapter 8

---

## REPL Workflow

---

Overview ...



---

# Chapter 9

---

## Trampolines

---

What is a trampoline? What problem do trampolines solve? How does one use Leiningen's trampoline facility?



---

# Chapter 10

---

## Testing

---

### Topics:

- [\*Unit Testing\*](#)
- [\*System Testing\*](#)
- [\*Continuous Integration\*](#)

Overview ...

## Unit Testing

---

Unit testing with Leiningen is easy! blah blah ...

### Unit Testing Concepts

Leiningen comes with a `test` command designed to work with unit testing frameworks. (?)

Here's a para ...



Attention: Pay attention!

### Unit Testing Procedure


Your code compiles.

Decide which unit testing framework you want to use. Clojure comes with its own framework (`Clojure.test`), but there are others available, e.g. [speclj](#) ("*speckle*"), [midje](#). A clojure plugin is available for midje.

This mini-tutorial covers only Clojure.test.

1. Do x by running `frob -x`  
You should see something like:

```
...screen dump here...
```

2.  Caution: Be sure you have safasfd before doing this!

Start out by xing using one of the following methods:.

- Using this method: `barfoo`
- Or this method: `bazfoo`

You should see something like:

```
...screen dump here...
```

3. Then this, which you do by:

- a) doing this
- b) and then this.

For example:

```
... sample code here...
```

4. And finally, this, which varies by platform:

Operating System	Step
On Linux	<code>foo.sh</code>
On Windows	<code>foo.bat</code>

Expected outcome of task as a whole...

Once you've finished this task, go on to ...

# System Testing

---

This topic blah blah...

## Sect Title

This is a great section!

## System Testing Concepts

Abstract...

General description of system testing task and problems...

## System Testing Procedure

Assumption: unit tests passed.

Some more info about setting up for this task...

1.  Caution:

Narrative here - `do foobar -baz`

- Using this method: `barfoo`
- Or this method: `bazfoo`

You should see something like:

```
...screen dump here...
```

2. Then this, which you do by:

- a) doing this
  - b) and then this.
- For example:

```
... sample code here...
```

3. And finally, this, which varies by platform:

Operating System	Step
On Linux	<code>foo.sh</code>
On Windows	<code>foo.bat</code>

Expected outcome of task as a whole...

Once you've finished this task, go on to ...

# Continuous Integration

---

This topic blah blah...

## Continuous Integration Concepts

Abstract...

Here's a para ...



Attention: Pay attention!

## Continuous Integration

Continuous integration using Jenkins

1. Caution:

Narrative here - `do foobar -baz`

- Using this method: `barfoo`
- Or this method: `bazfoo`

You should see something like:

```
...screen dump here...
```

2. Then this, which you do by:

- a) doing this
- b) and then this.

For example:

```
... sample code here...
```

3. And finally, this, which varies by platform:

Operating System

Step

On Linux

`foo.sh`

On Windows

`foo.bat`

Expected outcome of task as a whole...

Once you've finished this task, go on to ...



---

# Chapter 11

---

## Deployment

---

Overview ...



---

# Part

# II

---

## Extending Leiningen

---

### Topics:

- *Template Development*
  - *Plugin Development*
  - *Profile Development*
  - *Embedding Leiningen*
-



---

# Chapter 1

---

## Template Development

---

How to implement custom templates.

Should this go in a separate “Developer's Manual”?



---

# Chapter 2

---

## Plugin Development

---

How to implement custom tasks (plugins).  
Should this go in a separate “Developer's Manual”?





---

# Chapter 3

---

## Profile Development

---

How to implement custom profiles.

Should this go in a separate “Developer's Manual”?



---

# Chapter 4

---

## Embedding Leiningen

---

Leiningen is a Clojure library. You can use it from your own code.  
Should this go in a separate “Developer's Manual”?



---

# Appendix

---

## Appendices

---

Topics:

- [Contributing to this document](#)
-



---

# Appendix

# A

---

## Contributing to this document

---

WARNING: Document design can be hazardous to your other interests. (Apologies to Knuth)

This is a DITA document. The original is available from the [github project](#).

Please use the [Issues Tracker](#) to register bugs, corrections, enhancement requests, etc. Alternatively, you can clone the repository and edit the text directly. If you do please credit yourself in the metadata.

### Intellectual property

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

### DITA

[DITA](#) (Darwin Information Typing Architecture) is an [OASIS Standard](#) for writing, managing, and publishing information. It supports "topic-based" authoring.

The [DITA Wiki Knowledgebase](#) contains lots of information on DITA, from introductions and overviews to detailed documentation.

Another good overview is [DITA for the Impatient](#) from [XMLMind](#).

The [DITA Language Specification](#) documents the elements and attributes of the DITA Schema.

### Tools

There are two open source DITA transformation tools available:

- [DITA Open Toolkit](#)
- [XMLMind DITA Converter](#)





# Index

:base [30](#)  
*See also* Profiles  
 :dev [30](#)  
*See also* Profiles  
 :provided [30](#)  
*See also* Profiles  
 :system [30](#)  
*See also* Profiles  
 :user [30](#)  
*See also* Profiles

## A

Architecture [16](#)

## C

Classpath [36](#)  
 Command [16](#), [31](#)  
 Continuous integration [47](#)

## E

Effective project map [16](#)  
*See also* Map

## I

Infrastructure [16](#)  
 Integration, *See* Testing

## L

Leiningen kernel [16](#)

## M

Map [16](#)  
 Effective [16](#)  
 Project configuration [16](#)  
 semantics [16](#)

## N

New project [17](#)

## P

Profiles [16](#), [30](#)  
 system [30](#)  
 user-defined [30](#)  
 Project [16](#), [27](#), [29](#)  
 configuration map [16](#)  
 effective map , *See* Effective project map  
 map [16](#), [29](#)  
 map semantics [16](#)  
 structure [27](#)  
 Project configuration map , *See* Map

## R

Repository [17](#)

## S

Semantics [16](#)  
 Project map [16](#)  
 System testing, *See* Testing

## T

Testing [46-48](#)  
 continuous integration [48](#)  
 system [47](#)  
 unit [46](#)  
 Testing procedure [46-47](#)  
 system [47](#)  
 unit [46](#)

## U

Unit testing, *See* Testing

