Extracted from:

Clojure Applied

From Practice to Practitioner

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

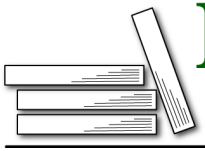
edited by Jacquelyn Carter

Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-074-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—April 8, 2015

We've now laid a solid foundation—representing our domain, building aggregate data, transforming data with functions, creating state, and using concurrency. It's time to start building larger units of code that correspond to the problem at hand. Separating our code into components allows us to think at a higher level in pieces that correspond to our problem. Component boundaries are also a good way to divide a code base between multiple developers or teams. They can also be opportunities for reuse.

We'll call these larger units *components*. Components are collections of finer-grained elements (functions, records, protocols) that have a greater overall purpose. They have an external API that callers will use. They also have an internal implementation, often including component state, and may even have use concurrency internally to process data in parallel or to create a separate thread of processing to react to events.

We'll start our consideration of components by looking at how to organize the functionality of our application into Clojure namespaces—this applies to all our code, both inside and outside components. Next we'll look at the external API that callers will use—this requires considering both the function call interface as well as use of longer-lived core.async channels. Finally we'll look at how to implement the component internals, managing component state and its lifecycle using the tools we've already seen for state and concurrency.

In the following chapter we'll be taking these components the next step to full application assembly.

Organizing With Namespaces

Clojure code is compiled and evaluated as a series of individual top-level forms (functions, records, protocols, etc.) but Clojure provides namespaces to group those individual forms. Namespaces are named, hierarchical containers that we can use to collect, organize, and name groups of forms. One practical use of namespaces is to allow us to use simple names in our code without worrying that we will conflict with the same name somewhere else—the namespace provides a means of specifying which one we mean.

While Clojure code is made up of finer-grained elements, dependencies are declared and loaded at the namespace level, not at the function level. The `ns` macro in each namespace defines its dependencies, collectively creating a dependency graph. Thus, namespaces also affect the order in which namespaces are loaded. In cases where a namespace provides an implementation of a multimethod or protocol (both open systems for type-specific behavior),

this load order may be important as implementations must be loaded before they can be used.

Both namespaces and components are tools for organization. Namespaces are a language feature for organizing functions whereas components are a means of organizing at the problem level. These two approaches are both useful and work in tandem to provide structure to our code, ultimately making it easier for other developers to understand and use.

Namespace Categories

There are many different reasons to group a set of functions into a namespace in Clojure. The following categories can be used to create a logical namespace architecture that reflects the application:

Utility Utility namespaces provide generic functions organized by domain or purpose. For example, you might create a namespace for string manipulation or parsing a particular file format. Generally utility namespaces have few dependencies.

Data Definition It is common to define either a custom collection or a set of domain entities in a namespace along with helper functions for using the collection or entities.

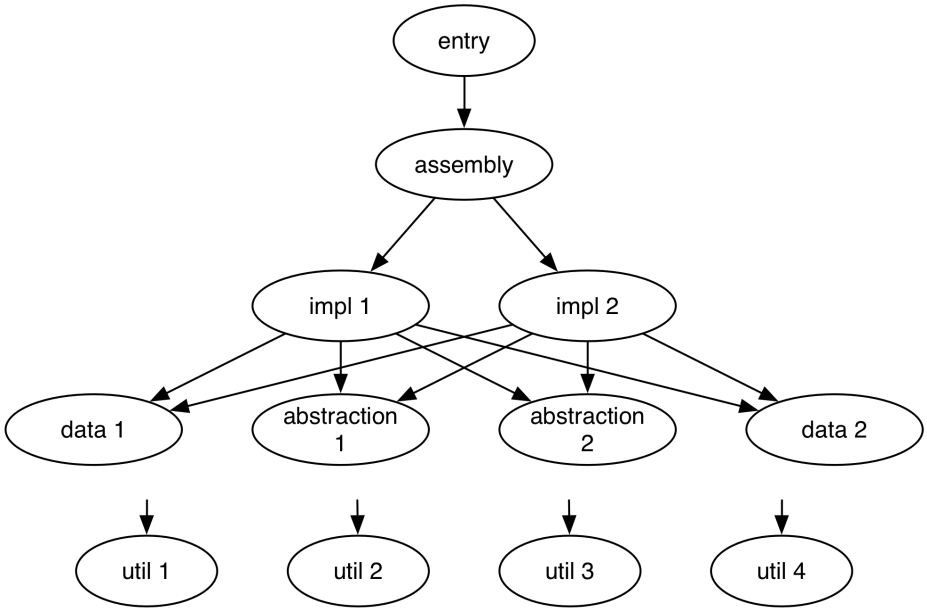
Abstraction Abstractions, like protocols, can be isolated in a namespace with minimal dependencies.

Implementation On the other side, it's often useful to implement an abstraction defined by a protocol or interface in a namespace. This implementation can then be assembled into an application.

Assembly Given a set of implementations and a configuration specifying how the implementations should be constructed and connected, an assembly namespace ties everything together. Inside the implementations, generally only the abstractions (protocols) or data structures will be used directly.

Entry point Most applications will have one or more entry points that connect the start of the application (which includes the gathering of configuration) to initiate assembly and other lifecycle operations.

The following diagram gives a view of how these kinds of namespaces typically layer together in a library or application.



This structure is a useful guideline in designing your own namespace structure. The utility namespaces are at the bottom of the dependency graph, with few or no dependencies of their own and in use by multiple namespaces above. The next layer consists of either data or abstraction namespaces, creating the building blocks for the application itself. Above the abstractions are the actual implementations of those abstractions. Above those you'll find an assembly layer where configuration is processed, implementations are assembled and connected, and application state is created. On the top are one or more entry points—web apps, command line interfaces, services, etc.

There are many approaches to organizing the namespaces in a project as a namespace tree and no one right answer. Smaller projects often place the majority of the namespaces within a single root named after the project with minimal nesting:

```
myproject.util.string  ;; utility
myproject.util.json    ;; utility
myproject.domain       ;; data - domain entities
myproject.config       ;; data - config data
myproject.services     ;; abstraction - service definitions
myproject.impl.xyz     ;; implementation of service abstraction
myproject.assembly     ;; assembly
myproject.main         ;; main entry point - command-line
```

For small systems, it's often easiest to cut services horizontally, grouping many abstractions, implementations, or utilities together. As your system grows, it will become increasingly useful to break your system into vertical slices where each particular component may consist of an API, an implementation, a set of data definitions, and utilities.

Public vs Private Functions

Clojure is biased towards making data and functions available by default. However, most namespaces have functions that are used as helpers or never intended to be part of the public usage. When you are defining the functions in a namespace, consider how a consumer will perceive those functions and how they're expected to use it. Some tools and conventions are private vars, documentation strings, and the namespace structure itself.

The primary tool built into Clojure itself is the ability to mark functions as private using `defn-` or the `^:private` meta tag:

```
(defn- internal-function [] ...)
(def ^:private internal-constant 42)
```

While these vars will be omitted from some namespace function results, they can still be accessed directly with the reader var syntax or by calling directly into the namespace object.

Some documentation-generation tools, like autodoc, will omit functions that do not have docstrings. Clojure core itself uses this feature to de-emphasize internal functions that are useful for advanced Clojure development, but not for general use.

Finally, it is common to see namespaces explicitly marked as being internal by using a namespace like `myproject.internal.db` where all namespaces under internal are considered non-public.

You may find any or all of these techniques useful in indicating to users of your own code where to start.

Now that we have some idea how to organize our namespaces, we should use those namespaces to create some components. We'll start by considering how to design the API of components before moving inside to how the components are implemented.

Designing Component APIs

When you identify a component within your application, you should begin by thinking about the purpose it will serve and how it will be used by other

components. Some typical kinds of components are *information managers*, *processors*, and *facades*. Information managers track state—either in-memory or in an external data store—providing operations to create, modify, or query that data. Processor components are all about data transformation or computation. Facade components exist primarily to make another external system accessible (and pluggable).

In reality, most components do not fit neatly into these boxes, but instead combine one or more aspects into a component that fulfills the unique needs of your own application.

The first thing to consider when designing a component is the API that outside consumers will use. There are two primary ways we can interact with components—invoking functions, and passing messages on a queue or channel. Let's look at functions first.

Manipulating Component Data With Functions

API functions are the knobs, buttons, or gauges on our component that allow an external consumer to interact with it. In Clojure there are a number of things that can be invoked as functions by a user but have different implementations—functions, macros, protocols, and multimethods. (There are others as well—maps, sets, keywords, symbols, etc but these are less useful as part of our API).

We've lifted our focus to a component, but we need to keep in mind everything we've learned so far. Whenever possible, components should expose immutable data directly. Due to immutability, there is no harm in handing back part of a component's data to the consumer—no copies are required and the component's own data is unaffected. Once the caller has the data, they are free to use all of the Clojure tools at their disposal in querying or transforming it.

Consider a knowledge engine component that manages a set of rules used for taking a request and formulating an automated response. Set aside the specific format of the rules for the moment, but assume each rule is defined as data. We'll need API functions to add, replace, and delete rules and a function to find rules based on some criteria. We'll also need some function to actually fire the rules and do the job at hand.

```
;; Read interface
(defn get-rules [ke])
(defn find-rules [ke criteria])

;; Update interface
(defn add-rule [ke rule])
```

```
(defn replace-rule [ke old-rule new-rule])
(defn delete-rule [ke rule])

;; Processing interface
(defn fire-rules [ke request])
```

We could then use these functions as follows:

```
(defn example []
  (let [ke (new-ke)]
    (add-rule ke :r1)
    (add-rule ke :r2)
    (add-rule ke :r3)
    (replace-rule ke :r1 :r1b)
    (delete-rule ke :r3)
    (get-rules ke)))
```

However, if we look a little deeper, we can see there are a smaller set of functions that can support the entire API:

```
;; Get the rule-set
(defn get-rules [ke])

;; Transform from one rule-set to another
(defn transform-rules [ke update-fn])

;; Produce a response from a request
(defn fire-rules [ke request])
```

The find-rules function can be implemented as a filter over get-rules. The add-rule, replace-rule, and delete-rule functions can all be seen as an application of transform-rules on the full rule-set.

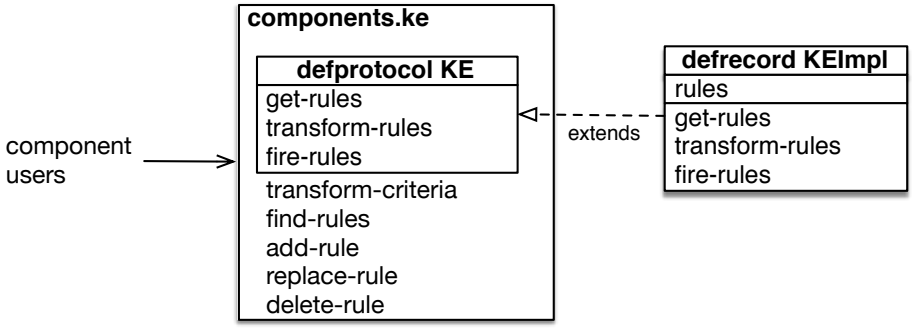
Most APIs have this pattern—a handful of key base functions and a larger set of functions provided for ease of use. Protocols are a good way to capture the core set of functions so that multiple implementations can extend that protocol. The derived functions should be provided in the API namespace and layered over the protocol. The API functions then work for any entity that extends the protocol.

Putting this all together in a full namespace would look like this:

```
(ns components.ke)

;; SPI protocol

(defprotocol KE
  (get-rules [ke] "Get full rule set")
  (transform-rules [ke update-fn]
    "Apply transformation function to rule set. Return new KE.")
```

```
  (fire-rules [ke request]
    "Fire the rules against the request and return a response"))

;; private helper functions

(defn- transform-criteria [criteria]
  ;; ...
  )

;; api fns built over the protocol

(defn find-rules
  [ke criteria]
  (filter (transform-criteria criteria) (get-rules ke)))

(defn add-rule
  [ke rule]
  (transform-rules ke #(conj % rule)))

(defn replace-rule
  [ke old-rule new-rule]
  (transform-rules ke #(-> % (disj old-rule) (conj new-rule))))

(defn delete-rule
  [ke rule]
  (transform-rules ke #(-> % (disj rule))))
```

This implementation defines a component API layered over a small extensible abstraction (the Service Provider Interface) as you can see in the following figure:



Creating a protocol for the entire API would require any implementation to reimplement all of the functions. Instead, the best tool in Clojure for collecting a set of related functions is the namespace, not the protocol. Protocols are best when defining a minimal abstraction for extension as we use here.

We'll come back to the state implementation parts of this component later, and continue focusing for now on other API considerations, like asynchronous calls.