Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional Clojure

Jeremy Anderson, Michael Gaare, Justin Holguín, Nick Bailey, Timothy Pratley

PROFESSIONAL CLOJURE

PROFESSIONAL

Clojure

Jeremy Anderson
Michael Gaare
Justin Holguín
Nick Bailey
Timothy Pratley

A Wiley Brand

Professional Clojure

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-119-26727-0
ISBN: 978-1-119-26728-7 (ebk)
ISBN: 978-1-119-26729-4 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

ABOUT THE AUTHORS

JEREMY ANDERSON is a developer at Code Adept, a West Michigan–based software consultancy focused on delivering high-quality software through providing software development, agile coaching, and training services. He is a Clojure enthusiast and contributor to a few different Clojure libraries. He is very passionate about teaching others how to program and volunteers to help teach computer science to area high-school and middle-school students.

MICHAEL GAARE is the platform technical lead at Nextangles, a financial technology startup. He's been using Clojure professionally since 2012 to build web services, data processing systems, and various libraries—not frameworks! In his spare time, he enjoys spending time with his wife and two daughters, and his hobby is opera singing.

JUSTIN HOLGUÍN is a software engineer at Puppet Labs, where he specializes in Clojure back-end services. Justin has a passion for functional programming and a special interest in technologies that improve software reliability, such as advanced type systems and property-based testing.

NICK BAILEY is a Clojure enthusiast and the maintainer of the Clojure java.jmx library. He is a software architect at DataStax, where he uses Clojure to build enterprise-level software for managing distributed databases. He was introduced to Clojure in 2010 and has been a fan ever since.

TIMOTHY PRATLEY is a Clojure contributor and advocate. Clojure has been his language of choice since 2008. He develops solutions in Clojure, ClojureScript, and Clojure-Android at his current role at Outpace Systems, Inc. He has 15 years of professional software development experience during which he has used many languages, frameworks, and databases. He loves Clojure, Datomic, pair programming, and thinking.

ABOUT THE TECHNICAL EDITORS

JUSTIN SMITH is a full-time Clojure developer who is active in the online Clojure community. His day job is 100% Clojure development.

ZUBAIR QURAISHI is a UX/Design and marketing hacker based in Denmark who has sold 2 startups and invested in over 30 startups over the last 20 years. He has been using Clojure and ClojureScript for the last 5 years. He has worked in many startups and Fortune 500 companies based in the United States, Europe, and Asia. You can find his blog is at `www.zubairquraishi.com`.

ALEX OTT is a software architect in Intel Security (formerly McAfee), based in Paderborn, Germany. He works in the area of information security and has been using Clojure since release 1.0 (2009) to build prototypes, internal services, and open source projects, like Incanter.

DOUG KNIGHT has been programming computers professionally for 18 years, using Microsoft technologies for most of that time. He switched to Ruby on Rails in 2014 when he joined LivingSocial, and in 2015 he added Clojure as part of his work for the company.

CREDITS

PROJECT EDITOR
Charlotte Kughen

TECHNICAL EDITOR
Justin Smith
Zubair Quraishi
Alex Ott
Doug Knight

PRODUCTION EDITOR
Barath Kumar Rajasekaran

COPY EDITOR
Troy Mott

MANAGER OF CONTENT DEVELOPMENT AND ASSEMBLY
Mary Beth Wakefield

PRODUCTION MANAGER
Kathleen Wisor

MARKETING MANAGER
Carrie Sherrill

PROFESSIONAL TECHNOLOGY & STRATEGY DIRECTOR
Barry Pruett

BUSINESS MANAGER
Amy Knies

EXECUTIVE EDITOR
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Nancy Bell

INDEXER
Nancy Guenther

COVER DESIGNER
Wiley

COVER IMAGE
©d8nn/Shutterstock

ACKNOWLEDGMENTS

JEREMY WOULD LIKE TO THANK God, first and foremost, for granting him the gifts that he has in order to do the things that he loves. Secondly, Jeremy thanks his family for being so supportive and understanding of him locking himself in his office to frantically write on evenings and weekends. Next, thanks to Christina Rudloff and Troy Mott for all the hard work they've done to put this project together and for inviting him onto this writing team, and thanks also to all the authors who helped make this project a reality. Finally, thanks to all the technical reviewers for taking the time to read and provide valuable feedback in the early stages of this book.

NICK WOULD LIKE TO THANK EVERYONE involved in making this book a reality: Troy and Christina for organizing, his fellow authors for writing and reviewing, and the technical reviewers for great feedback. He would also like to thank DataStax for giving him a chance to write Clojure professionally.

MICHAEL WOULD LIKE TO THANK LARA, Charlotte, and Juliette for their love, support, and understanding; Keith for his valuable assistance; Christina and Troy for their patience and the opportunity to write about a terrific subject; and Rich for creating something so interesting to write about—not to mention work with.

JUSTIN WOULD LIKE TO THANK HIS FAMILY for bootstrapping him and, among countless other things, encouraging his love of books and computers. He would also like to thank his many brilliant friends and colleagues at Puppet Labs, where he has been inspired and challenged to master Clojure bit by bit, day by day.

TIMOTHY WOULD LIKE TO THANK SHIN Nee for being the ultimate collaborator. He would like to thank you, the reader, for exploring how programming can be better; the Clojure community for providing a friendly, helpful, and pleasant ecosystem to exist in; and his parents for the many opportunities they crafted into his life.

CONTENTS

INTRODUCTION

WHAT IS CLOJURE?

> *Clojure is a dynamic, general-purpose programming language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language, yet remains completely dynamic—every feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection.*
>
> *Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures. When mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multithreaded designs.*
>
> —Rich Hickey, author of Clojure

This quote from Rich Hickey, the creator of Clojure, captures what Clojure is. Many people equate Clojure with functional programming, but much like Lisp, its predecessor, it's a general-purpose language that will support you no matter what paradigm you decide to program in.

Clojure is, however, very opinionated and offers great support for programming in a functional manner, with its focus on immutable values and persistent data structures. You may be surprised to know that Clojure also offers the ability to do object-oriented programming, which we cover in this book.

WHO IS THIS BOOK FOR?

This book was written with the professional programmer in mind. This means you should have experience programming in a language, and you should know the basic syntax and concepts in Clojure, and be ready to take Clojure programming to the next level. Our goal is to take you from a Clojure beginner to being able to think like a Clojure developer. Learning Clojure is much more than just learning a new syntax. You must use tools and constructs much differentely than anything you may be familiar with.

DEMO APPLICATION SOURCE CODE

You can access the source code from the Wiley website at `www.wiley.com/go/professionalclojure` or at our demo application via Github at `https://github.com/backstopmedia/clojurebook`.

A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes.

—Structure and Interpretation of Computer Programs

This book assumes some prior knowledge of Clojure and programming in general, but does not assume proficiency in Clojure. It will cover a broad scope of topics from changing the way you think and approach programming to how you integrate the REPL into your normal development routine to how you build real world applications using Ring and ClojureScript.

WHAT WILL YOU LEARN?

Our goal is to provide you with some real world examples of how to apply your Clojure knowledge to your day-to-day programming, not just theory and academia.

Chapter 1

In Chapter 1, you will learn about Clojure's unique view on designing programs. You'll discover some of the things that set Clojure apart from other languages, for example, how immutability is the default, and how Clojure qualifies as object-oriented programming.

Chapter 2

In Chapter 2, you will learn how to become proficient with the REPL and various tips and techniques for interacting with your actual application through the REPL. You'll learn how to run your code and tests from the REPL as well as how to write code that is easily reloaded from the REPL without having to restart it.

Chapter 3

In Chapter 3, you learn about building web services with Compojure, and the various concepts involved such as routes, handlers, and middleware. You will build a complete web service, and then learn various techniques for deploying your new application.

Chapter 4

Chapter 4 covers testing in Clojure, focusing primarily on the `clojure.test` testing library. You'll learn various techniques for many common testing scenarios, along with tools to help measure the quality of your code.

Chapter 5

In Chapter 5, you will learn how to build a task management web application similar to the popular Trello application in ClojureScript. You'll also learn the techniques for sharing functions between both your server-side and client-side applications.

Chapter 6

Chapter 6 takes a look at Datomic and how it applies the concept of immutability to databases. You'll learn the basics of how to model data in a Datomic database and how to extract that information. Then you'll apply this knowledge to building a database to support the task management application from Chapter 5.

Chapter 7

In Chapter 7, you'll take a look at performance and how to make your Clojure code execute faster. You'll discover how with a little work you can tweak your Clojure code to be as fast as Java code.

TOOLS YOU WILL NEED

Just as in any good adventure or journey, having the right tools makes things go much smoother. Fortunately, to work through the examples in this book, you only need three things: Java, Leiningen, and a good text editor.

Java

Most computers these days come with Java pre-installed, but in order to run the examples contained in this book you need to make sure you have installed a recent version. The code examples in this book were written with and confirmed to work with JDK 1.8.0_25. For instructions on how to download and install the proper JDK for your platform, see the documentation at Oracle's JDK download page: (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`).

Leiningen

Leiningen, according to their website (`http://leiningen.org`), is the most contributed-to Clojure project. For those of you coming from a background in Java, Leiningen fills a similar role that

Maven does for the Java world, only without all of the XML, and you can avoid wanting to pull your hair out. It helps you manage the dependencies for your project and declaratively describe your project and configuration, and provides access to a wealth of plugins for everything from code analysis to automation, and more. Leiningen makes your Clojure experience much more enjoyable.

Fortunately, getting Leiningen up and running is a fairly simple task. You'll want to install the latest version available, which at the time of this writing is 2.5.3. Please refer to the Leiningen website for instructions particular to your programming environment.

Editors

Once you have Leiningen installed, the only thing left to do is to make sure you have a good text editor to efficiently edit your Clojure code. If you have a favorite editor, just use what you're already comfortable with. However, if your editor doesn't support basic things like parentheses balancing, integration with the REPL, syntax highlighting, or properly indenting Clojure code, you may want to consider one of the editors below.

Emacs

Emacs is the favored editor of many grizzled veterans. It has a long history with Lisp. Even though it has a steep learning curve, it is considered by many to be very powerful, and no other editor is as extensible. There are many custom Emacs configurations designed to help ease the learning curve, such as Emacs Prelude (`https://github.com/bbatsov/prelude`), which also contains a sensible default configuration for developing in many languages, including Clojure.

LightTable

LightTable (`http://lighttable.com`) began life as a Kickstarter project with a unique new vision of how to integrate the code editor, REPL, and documentation browser for Clojure. It has delivered on those promises and then some and has gained popularity among many in the Clojure community.

Cursive (IntelliJ)

If you're already comfortable with using any of the various JetBrains IDEs, you'll be happy to know that there is a plugin for IntelliJ called Cursive (`https://cursive-ide.com`). Besides having good integration with nREPL, it also stays true to its reputation and contains excellent refactoring support, as well as debugging and Java interop.

Counterclockwise (Eclipse)

For those who are familiar with Eclipse, there is Counterclockwise (`http://doc.ccw-ide.org`), which can be installed as either an Eclipse plugin or a standalone product. Counterclockwise boasts many of the same features as the previous editors, integration with the REPL, and ability to evaluate code inline.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

> **NOTE** *Notes indicates notes, tips, hints, tricks, and/or asides to the current discussion.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show code within the text like so: `persistence.properties`.
- We show all code snippets in the book using this style:

```
FileSystem fs = FileSystem.get(URI.create(uri), conf);
InputStream in = null;
try {
```

- URLs in text appear like this: `http://<Slave Hostname>:50075`.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. All of the source code used in this book is available for download at `www.wiley.com`. Specifically for this book, the code download is on the Download Code tab at:

```
www.wiley.com/go/professionalclojure
```

You can also search for the book at `www.wrox.com` by ISBN (the ISBN for this book is 9781119267171 to find the code. And a complete list of code downloads for all current Wrox books is available at `www.wiley.com/dynamic/books/download.aspx`.

> **NOTE** *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-119-26727-0.*

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wiley.com/go/` and click the Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `http://p2p.wrox.com`. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com`, you will find a number of different forums that will help you, not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `http://p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

> **NOTE** *You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to This Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Have a Beginner's Mind

WHAT'S IN THIS CHAPTER?

- Understanding the differences between imperative and functional programming
- Learning how to think more functionally
- Discovering Clojure's unique perspective on object-oriented programming

> *If your mind is empty, it is always ready for anything, it is open to everything. In the beginner's mind there are many possibilities, but in the expert's mind there are few.*
>
> —Shunryu Suzuki

Over the past thirty years many popular programming languages have more in common with each other than they have differences. In fact, you could argue that once you have learned one language, it's not difficult to learn another. You merely have to master the subtle differences in syntax, and maybe understand a new feature that isn't present in the language that you're familiar with. It's not difficult to call yourself a polyglot programmer when many of the top languages in use today are all so similar.

Clojure, on the other hand, comes from a completely different lineage than most of the popular languages in use today. Clojure belongs to the Lisp family of programming languages, which has a very different syntax and programming style than the C-based languages you are probably familiar with. You must leave all of your programming preconceptions behind in order to gain the most from learning Clojure, or any Lisp language in general.

Forget everything you know, or think you know about programming, and instead approach it as if you were learning your very first programming language. Otherwise, you'll just be learning a new syntax, and your Clojure code will look more like Java/C/Ruby and less like Clojure is designed to look. Learning Clojure/Lisp will even affect the way you write in other languages, especially with Java 8 and Scala becoming more popular.

FUNCTIONAL THINKING

C, C++, C#, Java, Python, Ruby, and even to some extent Perl, all have very similar syntax. They make use of the same programming constructs and have an emphasis on an imperative style of programming. This is a style of programming well suited to the von Neumann architecture of computing that they were designed to execute in. This is probably most apparent in the C language, where you are responsible for allocating and de-allocating memory for variables, and dealing directly with pointers to memory locations. Other imperative languages attempt to hide this complexity with varying degrees of success.

> *In computer science, **imperative programming** is a **programming** paradigm that uses statements that change a program's state.*

This C-style of programming has dominated the programming scene for a very long time, because it fits well within the dominant hardware architectural paradigm. Programs are able to execute very efficiently, and also make efficient use of memory, which up until recently had been a very real constraint. This efficiency comes at the cost of having more complex semantics and syntax, and it is increasingly more difficult to reason about the execution, because it is so dependent upon the state of the memory at the time of execution. This makes doing concurrency incredibly difficult and error prone. In these days of cheap memory and an ever growing number of multiple core architectures, it is starting to show its age.

Functional programming, however, is based on mathematical concepts, rather than any given computing architecture. Clojure, in the spirit of Lisp, calls itself a general-purpose language; however, it does provide a number of functional features and supports the functional style of programming very well. Clojure as a language not only offers simpler semantics than its imperative predecessors, but it also has arguably a much simpler syntax. If you are not familiar with Lisp, reading and understanding Clojure code is going to take some practice. Because of its heavy focus on immutability, it makes concurrency simple and much less error prone than having to manually manage locks on memory and having to worry about multiple threads reading values simultaneously. Not only does Clojure provide all of these functional features, but it also performs object-oriented programming better than its Java counterpart.

Value Oriented

Clojure promotes a style of programming commonly called "value-oriented programming." Clojure's creator, Rich Hickey, isn't the first person to use that phrase to describe functional

programming, but he does an excellent job explaining it in a talk titled *The Value of Values* that he gave at Jax Conf in 2012 (`https://www.youtube.com/watch?v=-6BsiVyC1kM`).

By promoting this style of value-oriented programming, we are focused more on the values than mutable objects, which are merely abstractions of places in memory and their current state. Mutation belongs in comic books, and has no place in programming. This is extremely powerful, because it allows you to not have to concern yourself with worrying about who is accessing your data and when. Since you are not worried about what code is accessing your data, concurrency now becomes much more trivial than it ever was in any of the imperative languages.

One common practice when programming in an imperative language is to defensively make a copy of any object passed into a method to ensure that the data does not get altered while trying to use it. Another side effect of focusing on values and immutability is that this practice is no longer necessary. Imagine the amount of code you will no longer have to maintain because you'll be using Clojure.

In object-oriented programming, we are largely concerned with information hiding or restricting access to an object's data through encapsulation. Clojure removes the need for encapsulation because of its focus on dealing with values instead of mutable objects. The data becomes semantically transparent, removing the need for strict control over data. This level of transparency allows you to reason about the code, because you can now simplify complex functions using the substitution model for procedure application as shown in the following canonical example. Here we simplify a function called `sum-of-squares` through substituting the values:

```
(defn square [a] (* a a))
(defn sum-of-squares [a b] (+ (square a) (square b))

; evaluate the expression (sum-of-squares 4 5)

(sum-of-squares 4 5)
(+ (square 4) (square 5))
(+ (* 4 4) (* 5 5))
(+ 16 25)
41
```

By favoring functions that are referentially transparent, you can take advantage of a feature called memorization. You can tell Clojure to cache the value of some potentially expensive computation, resulting in faster execution. To illustrate this, we'll use the Fibonacci sequence, adapted for Clojure, as an example taken from the classic MIT text *Structure and Interpretation of Computer Programs (SICP)*.

```
(defn fib [n]
  (cond
    (= n 0) 0
    (= n 1) 1
    :else (+ (fib (- n 1))
             (fib (- n 2)))))
```