

LISP FOR THE WEB

BY ADAM TORNHILL



Lisp for the Web

Adam Tornhill

This book is for sale at <http://leanpub.com/lispweb>

This version was published on 2015-05-24



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Adam Tornhill

Tweet This Book!

Please help Adam Tornhill by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#lispforweb](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#lispforweb>

Contents

Introduction	1
What's new?	2
Source code to Lisp for the Web	2
Preparations	2
Get a Lisp	2
Installing libraries with Quicklisp	3
About Adam and Adam	3
About Adam Tornhill	3
Credits	3
Lisp for the Web	4
What to expect	4
The Lisp story	4
Crash course in Lisp	5
The Brothers are History	7
Representing Games	8
A prototypic backend	9
Customizing the printed representation of CLOS objects	11
Entering the Web	12
Generating HTML dynamically	12
Macros: Fighting the evils of code duplication	13
More than an opera	15
Meet the Hunchentoot web-server	15
Publishing content	15
Putting it together	17
Adding Games: forms and client input	19
Expressing JavaScript in Lisp	20
Lisp for the web browser	20
Generating JavaScript	20
On event handlers	21
Extending our DSL for custom scripts	22
Towards robustness	23
The Lisp advantage	24
Persistent Objects	25
MongoDB as a backend	25

CONTENTS

From Lisp to Mongo and back again	25
Migrating to persistence	26
Avoid Duplicates with Constraints	28
CLOS: Observers for free	28
Sorting games through MongoDB	30
Remembering the Games	30
MapReduce in Lisp	32
Pushing work to the server-side	32
The MapReduce algorithm in MongoDB	33
Specifying the steps with Parescript	34
Executing MapReduce from the REPL	34
A natural extension: presenting charts by category	36
Endgame	37
Final considerations	37
On backends	37
SQL	37
NoSQL alternatives	37
Persistent objects protocols	38
Moving on	39
Book recommendations	39
Code reading	39

Introduction

Lisp has kept me fascinated since I hesitantly opened my first pair of parenthesis. To this day, the lessons I took away from Lisp keep informing my decisions in other languages and technologies. Learning Lisp radically changed how I view programming.

But starting with Lisp wasn't a smooth transition. Lisp's power comes from a philosophy that's absent in all mainstream languages of today. In a way, learning Lisp was like facing programming as an absolute beginner again.

When it comes to learning, there's nothing more efficient than teaching someone else. Suddenly your cloudy thoughts need to be crystalized, put in context and communicated. In short, you need to truly understand your topic. Following that approach I set out to write an article on Lisp programming. I chose the web as my domain since my preliminary goal was to understand how a 50 year old language could provide just the right tools for an environment that wasn't even conceived back then. If Lisp indeed made it, there had to be something timeless over it. Some inherent quality that could give the programmers of today unequalled power and flexibility.

Writing the original Lisp for the Web article back in 2008 proved to be a rewarding experience. Not only did I succeed on this step in my learning journey; the feedback I got from fellow programmers around the world was overwhelming. I never thought I'd reach such a wide audience. In a way Lisp for the Web seemed to provide a natural next step for people that read Paul Graham's essays and were now looking for practical material to take their first step into the exciting (and sometimes frightening) world of Lisp.

Since that time back in 2008 there has been several additions to the Common Lisp ecosystem. Quicklisp lowered the hurdle by providing a simple way to get started, making all powerful libraries easy to install. New books turned up. In particular [Land of Lisp](#)¹ stands as an excellent example on a pedagogical introduction to the language. The web development community continued to evolve too. Just to mention a few examples, Edi Weitz made several improvements to his Hunchentoot web-server, NoSQL backends gained popularity, and Matthew Snyder expanded on my work with his excellent [Lisp for the Web, part II](#)² article.

To this day I keep getting questions on Lisp for the Web. Part of the reason is that I left the original code to rot. I moved on to pursuit other adventures and never really looked back. The result is that the code is now in a sad state where it isn't compatible with newer versions of Hunchentoot. That's not where I wanted it to end. I think the Common Lisp community deserves up-to-date material; I don't want anyone come to my tutorial with the intentions of learning Lisp and leave out of frustration as the examples don't compile.

¹<http://landoflisp.com>

²<http://msnyder.info/posts/2011/07/lisp-for-the-web-part-ii/>

What's new?

This book is my attempt to restore Lisp for the Web to its former glory. Besides bringing the code up-to-date, I've also taken this opportunity to switch to HTML5 and include some new material. This additional material includes:

- coverage of interacting with the popular NoSQL [MongoDB](http://www.mongodb.org)³,
- a treatment of the [MapReduce](http://en.wikipedia.org/wiki/MapReduce)⁴ algorithm in Lisp and how we can execute it on a remote node without ever leaving our Lisp environment, and
- expanded coverage on how to express JavaScript in Lisp and generate it dynamically, and
- finally, I dive deeper into the Common Lisp Object System (CLOS) and show how it allows us to model complex functionality with simple constructs. The example I include is a variant of the design pattern Observer, but in just one line of standard CLOS code rather than building elaborate class hierarchies.

In the end I hope you enjoy this book and find learning Lisp just as rewarding as I did.

Source code to Lisp for the Web

All source code is available in a [Github repository](https://github.com/adamtornhill/LispForTheWeb)⁵. The code comes in three versions: one with the prototypical backend we're about to develop in the first part of the book, another version that extends the code to use a persistent storage, and finally a module that includes the MapReduce implementation.

If you clone my Github repository you'll notice that I've duplicated several definitions in the different modules. This isn't something I necessarily encourage in production code. Rather my intent is to present an easily digestible, stand-alone snapshot of each version of the application. In the tutorial I take an iterative approach where I grow a live system organically. Often, this includes replacing existing functions with new versions. As such, the duplications are traces of what once was.

Preparations

Get a Lisp

I've organized this book as a tutorial. If you want to follow along you need a Lisp. There are several high-quality alternatives available to you, both open-source as well as commercial implementations. A good starting point is the Getting Started page on the [CLiki](http://www.cliki.net/Getting%20Started)⁶.

³<http://www.mongodb.org>

⁴<http://en.wikipedia.org/wiki/MapReduce>

⁵<https://github.com/adamtornhill/LispForTheWeb>

⁶<http://www.cliki.net/Getting%20Started>

Installing libraries with Quicklisp

Common Lisp has tons of interesting open-source libraries. We'll get to meet some of them in this book. Since the advent of [Quicklisp](http://www.quicklisp.org)⁷, installing a library is a breeze. You can get a copy of Quicklisp here: <http://www.quicklisp.org>⁸. Once you've downloaded your copy, just follow the instructions on the [Cliki](http://www.cliki.net)⁹ to install Quicklisp.

About Adam and Adam

The original Lisp for the Web article was published by an Adam Petersen. Now it may look as though I've stolen not only Mr Petersen's idea but also the bulk of his writings. Trust me, it's OK - I used to be him. I changed my family name in the summer of 2013 as I married my beautiful Jenny.

About Adam Tornhill

Adam is a programmer that combines degrees in engineering and psychology. He's the author of [Your Code as a Crime Scene](https://pragprog.com/book/atcrime/your-code-as-a-crime-scene)¹⁰, has written the popular Lisp for the Web tutorial and self-published a book on [Patterns in C](https://leanpub.com/patternsinc)¹¹. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

Credits

Thanks to Christopher Wellons, Stuart Malcolm and Matthew Malisz for reporting issues in earlier versions of this book.

My amazing wife [Jenny Tornhill](http://www.jennytornhill.se)¹² designed the cover of this book. Thanks Jenny - you rock!

⁷<http://www.quicklisp.org>

⁸<http://www.quicklisp.org>

⁹<http://www.cliki.net/Getting%20Started>

¹⁰<https://pragprog.com/book/atcrime/your-code-as-a-crime-scene>

¹¹<https://leanpub.com/patternsinc>

¹²<http://www.jennytornhill.se>

Lisp for the Web

With his essay [Beating the Averages](#)¹³, Paul Graham told the story of how his web start-up Viaweb outperformed its competitors by using Lisp. Lisp? Did I parse that correctly? That ancient language with all those scary parentheses? Yes, indeed! And with the goal of identifying its strengths and what they can do for us, I'll put Lisp to work developing a web application. In the process we'll find out how a 50 years old language can be so well-suited for modern web development and yes, it's related to all those parentheses.

What to expect

Starting from scratch, we'll develop a three-tier web application. I'll show how to:

- utilize powerful open source libraries for expressing dynamic HTML5 and JavaScript in Lisp,
- develop a small, embedded domain specific language tailored for my application,
- extend the typical development cycle by modifying code in a running system and execute code during compilation,
- migrate from data structures in memory to persistent objects using a third party NoSQL database (MongoDB), and
- finally show how we can execute a MapReduce algorithm on a remote database server without even leaving our Lisp environment.

I'll do this in a live system transparent to the users of the application. Because Lisp is so high-level, I'll be able to achieve everything in just around 80 lines of code.

This article will not teach you Common Lisp (for that purpose I provide some [book recommendations](#) at the end). Instead, I'll give a short overview of the language and try to explain the concepts as I introduce them, just enough to follow the code. The idea is to convey a feeling of how it is to develop in Lisp rather than focusing on the details.

The Lisp story

Lisp is actually a family of languages discovered by John McCarthy over 50 years ago. The characteristic of Lisp is that Lisp code is made out of Lisp data structures with the practical implication that it is not only natural, but also highly effective to write programs that write programs. This feature has allowed Lisp to adapt over the years. For example, as object-oriented programming became popular, powerful object systems could be implemented in Lisp as libraries

¹³<http://www.paulgraham.com/avg.html>

without any change to the core language. Later, the same proved to be true for aspect-oriented programming.

This idea is not only applicable to whole paradigms of programming. Its true strength lays in solving everyday problems. With Lisp, it's straightforward to build-up a domain specific language allowing us to program as close to the problem domain as our imagination allows. I'll illustrate the concept soon, but before we kick-off, let's look closer at the syntax of Lisp.

Crash course in Lisp

What Graham used for Viaweb was Common Lisp, an ANSI standardized language, which we'll use in this article too (the other main contenders are Scheme, generally considered cleaner and more elegant but with a much smaller library, and Clojure that introduces several interesting concepts and targets existing VMs).

Common Lisp is a high-level interactive language that may be either interpreted or compiled. You interact with Lisp through its top-level . The top-level is basically a prompt. On my system it looks like this:

```
CL-USER>
```

Through the top-level, we can enter expressions and see the results (the values returned by the top-level are highlighted):

```
CL-USER>(+ 1 2 3)  
6
```

As we see in the example, Lisp uses a prefix notation. A parenthesized expression is referred to as a form . When fed a form such as `(+ 1 2 3)` , Lisp generally treats the first element (+) as a function and the rest as arguments. The arguments are evaluated from left to right and may themselves be function calls:

```
CL-USER>(+ 1 2 (/ 6 2))  
6
```

We can define our own functions with *defun*:

```
CL-USER>(defun say-hello (to)  
  (format t "Hello, ~a" to))
```

Here we're defining a function *say-hello* , taking one argument: *to* . The *format* function is used to print a greeting and resembles a *printf* on steroids. Its first argument is the output stream and here we're using *t* as a shorthand for standard output. The second argument is a string, which in our case contains an embedded directive *~a* instructing format to consume one argument and output it in human-readable form. We can call our function like this:

```
CL-USER>(say-hello "ACCU")  
Hello, ACCU  
NIL
```

The first line is the side-effect, printing *“Hello, ACCU”* and *NIL* is the return value from our function. By default, Common Lisp returns the value of the last expression. From here we can redefine *say-hello* to return its greeting instead:

```
CL-USER>(defun say-hello (to)  
  (format nil "Hello, ~a" to))
```

With *nil* as its destination, *format* simply returns its resulting string:

```
CL-USER>(say-hello "ACCU")  
"Hello, ACCU"
```

Now we’ve gotten rid of the side-effect. Programming without side-effects is in the vein of functional programming, one of the paradigms supported by Lisp. Lisp is also dynamically typed. Thus, we can feed our function a number instead:

```
CL-USER>(say-hello 42)  
"Hello, 42"
```

In Lisp, functions are first-class citizens. That means, we can create them just like any other object and we can pass them as arguments to other functions. Such functions taking functions as arguments are called higher-order functions. One example is *mapcar*. *mapcar* takes a function as its first argument and applies it subsequently to the elements of one or more given lists:

```
CL-USER>(mapcar #'say-hello (list "ACCU" 42 "Adam"))  
("Hello, ACCU" "Hello, 42" "Hello, Adam")
```

The funny *#’* is just a shortcut for getting at the function object. As you see above, *mapcar* collects the result of each function call into a list, which is its return value. This return value may of course serve as argument to yet another function:

```
CL-USER>(sort (mapcar #'say-hello (list "ACCU" 42 "Adam")) #'string-lessp)  
("Hello, 42" "Hello, ACCU" "Hello, Adam")
```

Lisp itself isn’t hard, although it may take some time to wrap ones mindset around the functional style of programming. As you see, Lisp expressions are best read inside-out. But the real secret to understanding Lisp syntax is to realize that the language doesn’t have one; what we’ve been entering above is basically parse-trees, generated by compilers in other languages. And, as we’ll see soon, exactly this feature makes it suitable for metaprogramming.

The Brothers are History

Remember the hot gaming discussions 20 years ago? “Giana Sisters” really was way better than “Super Mario Bros”, wasn’t it? We’ll delegate the question to the wise crowd by developing a web application. Our web application will allow users to add and vote for their favourite retro games. A screenshot of the end result is provided in Figure 1 below.



The Retro Games front page

From now on, I start to persist my Lisp code in textfiles instead of just entering expressions into the top-level. Further, I define a package for my code. Packages are similar to namespaces in C++ or Java’s packages and helps to prevent name collisions (the main distinction is that packages in Common Lisp are first-class objects).

```
(defpackage :retro-games
  (:use :cl :cl-who :hunchentoot :parenscrip))
```

The new package is named `:retro-games` and I also specify other packages that we’ll use initially:

- CL is Common Lisp’s standard package containing the whole language.
- [CL-WHO](http://weitz.de/cl-who/)¹⁴ is a library for converting Lisp expressions into XHTML.
- [Hunchentoot](http://weitz.de/hunchentoot/)¹⁵ is a web-server, written in Common Lisp itself, and provides a toolkit for building dynamic web sites.
- [Parenscrip](http://common-lisp.net/project/parenscrip/)¹⁶ allows us to compile Lisp expressions into JavaScript. We’ll use this for client-side validation.

¹⁴<http://weitz.de/cl-who/>

¹⁵<http://weitz.de/hunchentoot/>

¹⁶<http://common-lisp.net/project/parenscrip/>

Using Quicklisp, installing these libraries is as simple as it gets:

```
CL-USER> (ql:quickload '(cl-who hunchentoot parenscrip))
```

The *quickload* function will ensure that the listed libraries are properly installed, transparently resolving and installing their dependencies.

With my package definition in place, I'll put the rest of the code inside it by switching to the *:retro-games* package:

```
(in-package :retro-games)
```

Most top levels indicate the current package in their prompt. On my system the prompt now looks like this:

```
RETRO-GAMES>
```

Representing Games

With the package in place, we can return to the problem. It seems to require some representation of a game and I'll choose to abstract it as a class:

```
(defclass game ()  
  ((name :initarg :name)  
   (votes :initform 0)))
```

The expression above defines the class *game* without any user-specified superclasses, hence the empty list *()* as second argument. A game has two slots (slots are similar to attributes or members in other languages): a *name* and the number of accumulated *votes*. To create a game object I invoke *make-instance* and pass it the name of the class to instantiate:

```
RETRO-GAMES> (defvar many-lost-hours (make-instance 'game :name "Tetris"))  
MANY-LOST-HOURS
```

Because I specified an initial argument in my definition of the *name* slot, I can pass this argument directly and initialize that slot to *"Tetris"*. The *votes* slot doesn't have an initial argument. Instead I specify the code I want to run during instantiation to compute its initial value through *:initform*. In this case the code is trivial, as I only want to initialize the number of votes to zero. Further, I use *defvar* to assign the object created by *make-instance* to the variable *many-lost-hours*.

Now that we got an instance of *game* we would like to do something with it. We could of course write code ourselves to access the slots. However, there's a more lispy way; *defclass* provides the possibility to automatically generate accessor functions for our slots:

```
(defclass game ()
  ((name :reader name
        :initarg :name)
   (votes :accessor votes
          :initform 0)))
```

The option *:reader* in the name slot will automatically create a read function and the option *:accessor* used for the votes slot will create both read and write functions. Lisp is pleasantly uniform in its syntax and these generated functions are invoked just like any other function:

```
RETRO-GAMES>(name many-lost-hours)
"Tetris"
RETRO-GAMES>(votes many-lost-hours)
0
RETRO-GAMES>(incf (votes many-lost-hours))
1
RETRO-GAMES>(votes many-lost-hours)
1
```

The only new function here is *incf*, which when given one argument increases its value by one. We can encapsulate this mechanism in a method used to vote for the given game:

```
(defmethod vote-for (user-selected-game)
  (incf (votes user-selected-game)))
```

The top-level allows us to immediately try it out and vote for Tetris:

```
RETRO-GAMES>(votes many-lost-hours)
1
RETRO-GAMES>(vote-for many-lost-hours)
2
RETRO-GAMES>(votes many-lost-hours)
2
```

A prototypic backend

Before we can jump into the joys of generating web pages, we need a backend for our application. Because Lisp makes it so easy to modify existing applications, it's common to start out really simple and let the design evolve as we learn more about the problem we're trying to solve. Thus, I'll start by using a list in memory as a simple, non-persistent storage.

```
(defvar *games* '())
```

The expression above defines and initializes the global variable (actually the Lisp term is special variable) **games** to an empty list. The asterisks aren't part of the syntax; it's just a naming convention for globals. Lists may not be the most efficient data structure for all problems, but Common Lisp has great support for lists and they are easy to work with. Later we'll change to a real database and, with that in mind, I encapsulate the access to **games** in some small functions:

```
(defun game-from-name (name)
  (find name *games* :test #'string-equal
        :key #'name))
```

Our first function *game-from-name* is implemented in terms of *find*. *find* takes an item and a sequence. Because we're comparing strings I tell *find* to use the function *string-equal* for comparison (remember, *#'* is a short cut to refer to a function). I also specify the key to compare. In this case, we're interested in comparing the value returned by the *name* function on each game object.

If there's no match *find* returns *NIL*, which evaluates to *false* in a boolean context. That means we can reuse *game-from-name* when we want to know if a game is stored in the **games** list. However, we want to be clear with our intent:

```
(defun game-stored? (game-name)
  (game-from-name game-name))
```

As illustrated in Figure 1, we want to present the games sorted on popularity. Using Common Lisp's *sort* function this is pretty straightforward; we only have to take care, because for efficiency reasons *sort* is destructive. That is, *sort* is allowed to modify its argument. We can preserve our **games** list by passing a copy to *sort*. I tell *sort* to return a list sorted in descending order based on the value returned by the *votes* function invoked on each game:

```
(defun games ()
  (sort (copy-list *games*) #'> :key #'votes))
```

So far the queries. Let's define one more utility for actually adding games to our storage:

```
(defun add-game (name)
  (unless (game-stored? name)
    (push (make-instance 'game :name name) *games*)))
```

push is a modifying operation and it prepends the game instantiated by *make-instance* to the **games** list. Let's try it all out at the top level.

```
RETRO-GAMES>(games)
NIL
RETRO-GAMES>(add-game "Tetris")
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(game-from-name "Tetris")
#<GAME @ #x71b943c2>
RETRO-GAMES>(add-game "Tetris")
NIL
RETRO-GAMES>(games)
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(mapcar #'name (games))
("Tetris")
```

Customizing the printed representation of CLOS objects

The values returned to the top level may not look too informative. It's basically the printed representation of a game object. Common Lisp allows us to customize how an object shall be printed. That's done by specializing the generic function *print-object* for our *game* class:

```
(defmethod print-object ((object game) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (name votes) object
      (format stream "name: ~s with ~d votes" name votes))))
```

With this specialisation in place, our game objects now look more informative in the REPL:

```
#<GAME name: "Tetris" with 2 votes>
```

The *print-object* specialisation introduces a few utilities from the standard library:

- *print-unreadable-object* helps us with the output stream set-up and displays type information in the printout.
- *with-slots* lets us reference the slots in the *game* instance as though they were variables.

Particularly *with-slots* is an example of how Lisp is grown to avoid all signs of duplication, no matter how subtle they may be (without *with-slots* we would have to access the *game* object twice in *format*).

We'll soon learn how to grow our own extensions to Lisp. But first, with this prototypic backend in place, we're prepared to enter the web.

Entering the Web

Generating HTML dynamically

The first step in designing an embedded domain specific language is to find a Lisp representation of the target language. For HTML this is really simple as both HTML and Lisp are represented in tree structures, although Lisp is less verbose. Here's an example using the CL-WHO library:

```
(with-html-output (*standard-output* nil :indent t)
  (:html
    (:head
      (:title "Test page"))
    (:body
      (:p "CL-WHO is really easy to use")))))
```

This code will expand into the following HTML, which is outputted to **standard-output**:

```
<html>
  <head>
    <title>Test page </title>
  </head>
  <body>
    <p> CL-WHO is really easy to use </p>
  </body>
</html>
```

HTML5 is more minimalistic and less verbose than its predecessors. But we still need to start a document by specifying its *doctype*. CL-WHO will do the job for us if we just tell it to. We do that by requesting a *prologue*. Since CL-WHO supports multiple markup types, we also need to specify that we want HTML5:

```
(setf (html-mode) :html5) ; output in HTML5 from now on

(with-html-output (*standard-output* nil :prologue t :indent t)
  (:html
    (:head
      (:title "Test page"))
    (:body
      (:p "CL-WHO is really easy to use")))))
```

With the *prologue* argument, our code now expands into the following valid HTML5:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Test page
  </title>
  </head>
  <body>
    <p>CL-WHO is really easy to use
  </p>
  </body>
</html>

```

CL-WHO also allows us to embed Lisp expressions, setting the scene for dynamic web pages.

Macros: Fighting the evils of code duplication

Although CL-WHO does provide a tighter representation than raw HTML we're still facing the potential risk of code duplication; the *html*, *head*, and *body* tags form a pattern that will recur on all pages. And it'll only get worse as we start to include additional content where we have to include more tags and attributes and, of course, start every page with that funny DOCTYPE line.

Further, if you look at Figure 1 you'll notice that the retro games page has a header with a picture of that lovely Commodore 64 and a strap line. I want to be able to define that header once and have all my pages using it automatically. The problem screams for a suitable abstraction and this is where Lisp differs from other languages. In Lisp, we can actually take on the role of a language designer and extend the language with our own syntax. The feature that allows this is macros. Syntactically, macros look like functions, but are entirely different beasts. Sure, just like functions macros take arguments. The difference is that the arguments to macros are source code, because macros are used by the compiler to generate code.

Macros can be a conceptual challenge as they erase the line between compile time and runtime. What macros do are expanding themselves into code that are actually compiled. In their expansion macros have access to the whole language, including other macros, and may call functions, create objects, etc.

So, let's put this amazing macro mechanism to work by defining a new syntactic construct, the `standard-page`. A `standard-page` will abstract away all HTML5 boiler plate code and automatically generate the heading on each page. The macro will take two arguments. The first is the title of the page and the second the code defining the body of the actual web-page. Here's a simple usage example:

```

(standard-page (:title "Retro Games")
  (:h1 "Top Retro Games")
  (:p "We'll write the code later..."))

```

Much of the macro will be straightforward CL-WHO constructs. *standard-page* will take two arguments: a keyword *:title* and a *body* argument that wraps-up the rest of the provided code. Using the backquote syntax (the ‘ character), we can specify a template for the code we want to generate:

```

1 (defmacro standard-page ((&key title) &body body)
2   `(with-html-output-to-string
3     (*standard-output* nil :prologue t :indent t)
4     (:html :lang "en"
5       (:head
6         (:meta :charset "utf-8")
7         (:title ,title)
8         (:link :type "text/css"
9               :rel "stylesheet"
10              :href "/retro.css"))
11       (:body
12         (:div :id "header" ; Retro games header
13           (:img :src "/logo.jpg"
14                 :alt "Commodore 64"
15                 :class "logo")
16           (:span :class "strapline"
17                 "Vote on your favourite Retro Game"))
18         ,@body))))

```

Within the backquoted expression we can use , (comma) to evaluate an argument and ,@ (comma-at) to evaluate and splice a list argument. Remember, the arguments to a macro are code. In this example the first argument *title* is bound to “Retro Games” and the second argument *body* contains the *:h1* and *:p* expressions wrapped-up in a list. In the macro definition, the code bound to these arguments is simply inserted on the proper places in our backquoted template code (*title* on line 7, the *body* at line 18).

The power we get from macros become evident as we look at the generated code. The three lines in the usage example above expands into this (note that Lisp symbols are case-insensitive and thus usually presented in uppercase):

```

(WITH-HTML-OUTPUT-TO-STRING (*STANDARD-OUTPUT* NIL :PROLOGUE T :INDENT T)
  (:HTML :LANG "en"
    (:HEAD
      (:META :CHARSET "utf-8")
      (:TITLE "Retro Games")
      (:LINK :TYPE "text/css"
              :REL "stylesheet"
              :HREF "/retro.css"))
    (:BODY
      (:DIV :ID "header"
        (:IMG :SRC "/logo.jpg"

```

```

:ALT "Commodore 64"
:CLASS "logo")
(:SPAN :CLASS "strapline" "Vote on your favourite Retro Game"))
(:H1 "Top Retro Games")
(:P "We'll write the code later..."))))

```

This is a big win; all this is code that we don't have to write. Now that we have a concise way to express web-pages with a uniform look, it's time to introduce Hunchentoot.

More than an opera

Meet the Hunchentoot web-server

Named after a Zappa sci-fi opera, Edi Weitz's Hunchentoot is a full featured web-server written in Common Lisp. To launch Hunchentoot, we first have to make an instance of one of its provided acceptors. This object is, as its name suggests, responsible for accepting incoming connections. Hunchentoot includes a few acceptors out of the box. We're gonna use an instance of the *easy-acceptor* class (other examples include the *ssl-acceptor* for https servers). It's pre-fixed "easy" since the acceptor automatically provides a dispatch mechanism for us. We'll discuss the dispatch mechanism soon. Just one more thing: in order to start the server, we invoke the *start* method on the created *easy-acceptor* instance:

```
RETRO-GAMES>(start (make-instance 'easy-acceptor :port 8080))
```

start supports several arguments, but we're only interested in specifying a port other than the default port 80. To make the code a bit more expressive I just wrap the functionality before invoking the function:

```

(defun start-server (port)
  (start (make-instance 'easy-acceptor :port port)))

```

```
RETRO-GAMES> (start-server 8080)
```

And that's it - the server's up and running. We can test it by pointing a web browser to <http://localhost:8080/>, which should display Hunchentoot's default page.

Publishing content

To actually publish something, we have to provide Hunchentoot with a handler. In Hunchentoot all requests are dynamically dispatched to an associated handler. Its framework contains several functions for defining dispatchers. The code below creates a dispatcher and adds it to Hunchentoot's dispatch table:

```
(push (create-prefix-dispatcher "/retro-games.htm"
                                'retro-games)
      *dispatch-table*)
```

The dispatcher will invoke the function, *retro-games*, whenever an URI request starts with */retro-games*. Now we just have to define the *retro-games* function that generates the HTML:

```
(defun retro-games ()
  (standard-page (:title "Retro Games")
                 (:h1 "Top Retro Games")
                 (:p "We'll write the code later...")))
```

That's it - the retro games page is online. But I wouldn't be quick to celebrate; while we took care to abstract away repetitive patterns in *standard-page*, we've just run into another more subtle form of duplication. The problem is that every time we want to create a new page we have to explicitly create a dispatcher for our handle. Wouldn't it be nice if Lisp could do that automatically for us? Basically I want to be able to define a function just as I would with *defun*. The only difference is that this function would automatically have Lisp to create a handler, associate it with a dispatcher and put it in the dispatch table as I compile the code. Guess what, using macros the syntax is ours. And we don't even have to write them ourselves.

Since I wrote the initial Lisp for the Web article, Hunchentoot has been extended with some quite convenient macros. One of them is *define-easy-handler* that does just what we want:

```
(define-easy-handler (retro-games :uri "/retro-games") ()
  (standard-page (:title "Retro Games")
                 (:h1 "Top Retro Games")
                 (:p "We'll write the code later...")))
```

Now our “wish code” above actually compiles and generates the following Lisp code:

```
(PROGN
  (LET ((#:URI906 "/retro-games"))
    (PROGN
      (SETQ HUNCHENTOOT::*EASY-HANDLER-ALIST*
            (DELETE-IF
              (LAMBDA (LIST)
                (AND
                  (OR (EQUAL #:URI906 (FIRST LIST))
                      (EQ 'RETRO-GAMES (THIRD LIST)))
                  (OR (EQ T T) (INTERSECTION T (SECOND LIST))))))
            HUNCHENTOOT::*EASY-HANDLER-ALIST*))
      (PUSH (LIST #:URI906 T 'RETRO-GAMES)
            HUNCHENTOOT::*EASY-HANDLER-ALIST*)))
  (DEFUN RETRO-GAMES (&KEY)
    (STANDARD-PAGE (:TITLE "Retro Games")
                   (:H1 "Top Retro Games")
                   (:P "We'll write the code later..."))))
```

The code looks pretty compact at first since it includes several constructs we haven't discussed yet. But stay with me; since we have it all encapsulated in a ready-made macro we can safely ignore the bulk of the code for now. Just observe how the first part sets-up a handler and registers it with an URI (*/retro-games*). We also see that *define-easy-handler* defines a *defun retro-games* function for us. That's the function that will now automatically be invoked as a client requests the */retro-games* URI.

Leaving the details aside, there's a few interesting things about this macro:

- It illustrates that macros can take other macros as arguments. The Lisp compiler will continue to expand the macros and *standard-page* will be expanded too, writing even more code for us.
- Macros may execute code as they expand. If we dig into the source code, we'll see that the macro assembles our *defun retro-games* function by investigating the arguments we provide. The intent is to let us specify any *GET* or *POST* parameters that we expect. We don't use that functionality yet, but it will come in handy soon. For now, just note that's there's no trace of it in the generated code - just the pure function definition.
- Finally, a macro must expand into a single form, but we actually need two forms; a function definition and the code for creating a dispatcher. *progn* solves this problem by wrapping the forms in a single form and then evaluating them in order.

Putting it together

Phew, that was a lot of Lisp in a short time. But using the abstractions we've created, we're able to throw together the application in no time. Let's code out the main page as it looks in Figure 1 above:

```
(define-easy-handler (retro-games :uri "/retro-games") ()
  (standard-page
    (:title "Top Retro Games")
    (:h1 "Vote on your all time favourite retro games!")
    (:p "Missing a game? Make it available for votes "
      (:a :href "new-game" "here"))
    (:h2 "Current stand")
    (:div :id "chart" ; Used for CSS styling of the links.
      (:ol
        (dolist (game (games))
          (htm
            (:li (:a :href (format nil "vote?name=~a"
              (url-encode ; avoid injection attacks
                (name game)))) "Vote!")
            (fmt "~A with ~d votes" (escape-string (name game))
              (votes game))))))))))
```

Here we utilize our freshly developed embedded domain specific language (DSL) for creating *standard-pages*, a macro that we use together with the mini-DSL *define-easy-handler* from

Hunchentoot. The subsequent lines are straightforward HTML generation, including a link to *new-game*; a page we haven't specified yet. We will use some CSS to style the *Vote!* links to look and feel like buttons, which is why I wrap the list in a *div*-tag.

The first embedded Lisp code is *dolist*. We use it to create each game item in the ordered HTML list. *dolist* works by iterating over a list, in this case the return value from the *games*-function, subsequently binding each element to the *game* variable. Using *format* and the access methods on the *game* object, I assemble the presentation and a destination for *Vote!*. Here's some sample HTML output from one session:

```
<div id='chart'>
  <ol>
    <li>
      <a href='vote?name=Super Mario Bros'>Vote!</a>
      Super Mario Bros with 12 votes
    </li>
    <li>
      <a href='vote?name=Last Ninja'>Vote!</a>
      Last Ninja with 11 votes
    </li>
  </ol>
</div>
```

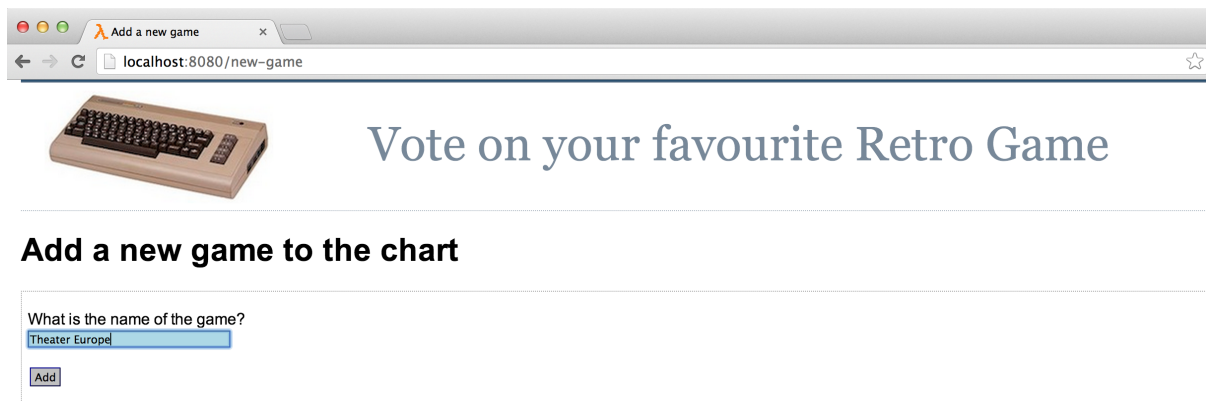
As the user presses *Vote!* we'll get a request for the *vote* URL with the name of the game attached as a query parameter. And now our usage of *define-easy-handler* starts to pay-off rapidly. As discussed earlier, the macro includes functionality to automatically extract the parameters from the http request. We just specify our parameters as we normally would with function arguments:

```
(define-easy-handler (vote :uri "/vote") (name)
  (when (game-stored? name)
    (vote-for (game-from-name name)))
  (redirect "/retro-games"))
```

When invoked, Hunchentoot will extract the *name* parameter from the URL and bind its value to the *name* parameter of our function. We pass this value to our back end abstraction to check if the *name* matches a *game-stored?*. *when* it does, we just instantiate a *game* object via our *game-from-name* function and *vote-for* it.

After a *vote-for* the requested game, Hunchentoot's *redirect* function takes the client back to the updated chart.

Adding Games: forms and client input



Making a new game available for voting

Now when we're able to vote we need some games to *vote-for*. In the code for the *retro-games* page above, I included a link to *new-game*. That page is displayed in Figure 2. Basically it contains a HTML form with a text input for the game name:

```
(define-easy-handler (new-game :uri "/new-game") ()
  (standard-page (:title "Add a new game")
    (:h1 "Add a new game to the chart")
    (:form :action "/game-added" :method "post" :id "addform"
      (:p "What is the name of the game?" (:br)
        (:input :type "text" :name "name" :class "txt"))
      (:p (:input :type "submit" :value "Add" :class "btn")))))
```

As the user submits the form, its data is dispatched to *game-added*:

```
(define-easy-handler (game-added :uri "/game-added") (name)
  (unless (or (null name) (zerop (length name)))
    (add-game name))
  (redirect "/retro-games"))
```

The first line in our easy handler definition should look familiar; just as in our *vote* function, we let our DSL extract the value of the name parameter and binds it to a local variable (*name*). Here we have to guard against an empty name. After all, there's nothing forcing the user to write anything into the field before submitting the form (we'll see in a minute how to add client-side validation). If we get a valid *name*, we add it to our database through the *add-game* function.

Expressing JavaScript in Lisp

Lisp for the web browser

Our *game-added* functions ensures that the user entered a valid game name. But at that point in time we're already on server side. Preferably we want to ensure that the user at least typed something before submitting the form. Can we do that in Lisp? Yes, actually. We can write Lisp code that compiles into JavaScript and we use the Parescript library for the task. Let's have a go at a validation function:

```
(defun validate-game-name (evt)
  (when (= (@ add-form name value) "")
    (chain evt (prevent-default))
    (alert "Please enter a name.")))
```

The *validate-game-name* is defined like any other Lisp function. JavaScript has quite different semantics than Common Lisp though. Using some convenience macros and utilities from Parescript we can bridge parts of the semantic gap and express our intent in a direct way:

- The *@* macro: This macro is used to access object properties. Our usage (*@ add-form name value*) compiles to the following JavaScript: *addForm.name.value*;
- The *chain* macro: This is a convenient way to chain together function calls on an object instance. In the code above, (*chain evt (prevent-default)*) will compile to *evt.preventDefault()*;

There are a few general notes of interest. First, Common Lisp and JavaScript have different naming conventions. That's why a Lisp symbol like *add-form* compiles to *addForm* in the JavaScript world. Similarly, comments in Lisp (indicated by *;*) will be preserved in the generated JavaScript but now expressed using the */*** code comment notation. Second, while JavaScript has a core that's well-suited for functional programming, many of its usages are about either validation or manipulating the DOM. As such, you'll often find that your Parescript code is richer on side-effects.

Generating JavaScript

To actually generate JavaScript code we need to wrap our definitions in the *ps* construct from Parescript. Using *ps*, the *defun* above expands into the following JavaScript:

```
function validateGameName(evt) {
  if (addForm.name.value === '') {
    evt.preventDefault();
    return alert('Please enter a name.');
```

This will do the trick. When invoked, the generated *validateGameName* will check to see if we've provided at least some text in the input field. If not, we signal a failed validation through the event API (*preventDefault*) and present a message to the user through a popup box (*alert*). To make it work, we just need to hook into JavaScript's event handling mechanism.

On event handlers

In my original article, I embedded my event handler directly in the form:

```
; inline JavaScript - do not try this at home:
(:form :action "/game-added"
  :method "post"
  :onsubmit
  (ps-inline
    (when (= name.value "")
      (alert "Please enter a name.")
      (return false))))
```

For the simplest of event handlers this may actually be fine. But unobtrusive JavaScript is an important design principle for a reason and Parescript supports that style too. What we really want to do is to keep the markup separated from the functions operating on the DOM.

One way to hook into the event mechanism is to dynamically add our validation function to the *onsubmit* event as the complete content (including CSS and our images) has been loaded. The *onload* event gives us that guarantee. To avoid clutter, I encapsulate the event subscription in a separate function (*init*) and register it with *onload*:

```
(defvar add-form nil)

(defun init ()
  (setf add-form (chain document (get-element-by-id "addform")))
  (chain add-form (add-event-listener "submit"
    validate-game-name false)))
(setf (chain window onload) init)))
```

I want to add the event handler to the form used for input. As visible in the code above, I access the form by its ID (*get-element-by-id*) so obviously we must assign that ID to the form:

```
(:form :action "/game-added" :method "post" :id "addform"
  ...code as before..
```

Putting it all together, our validation code expressed in Lisp will compile into the following JavaScript:

```
var addForm = null;

function validateGameName(evt) {
  if (addForm.name.value === '') {
    evt.preventDefault();
    return alert('Please enter a name.');
```

```
  };
};

function init() {
  addForm = document.getElementById('addform');
  return addForm.addEventListener('submit', validateGameName, false);
};

window.onload = init;
```

Extending our DSL for custom scripts

So, we've seen how we can generate validation functions for the web browser. But how do we integrate the validation with our small DSL that we developed earlier? I'd say we need to extend it to make it simple to provide scripting content in a *standard-page* definition. Not all pages will need scripting capabilities. Thus I make the script optional in the macro re-definition:

```
(defmacro standard-page ((&key title script) &body body)
  `(with-html-output-to-string
    (*standard-output* nil :prologue t :indent t)
    (:html :lang "en"
      (:head
        (:meta :charset "utf-8")
        (:title ,title)
        (:link :type "text/css"
          :rel "stylesheet"
          :href "/retro.css")
        ,(when script
          `(:script :type "text/javascript"
            (str ,script))))
      (:body
        (:div :id "header" ; Retro games header
          (:img :src "/logo.jpg"
```

```

      :alt "Commodore 64"
      :class "logo")
    (:span :class "strapline"
      "Vote on your favourite Retro Game"))
    ,@body))))

```

The API now accepts an additional keyword parameter, *script*. When present, the form associated with *script* will expand into Parenscript code in *standard-page*. The complete usage example in the */new-game* handler now looks like this:

```

(define-easy-handler (new-game :uri "/new-game") ()
  (standard-page (:title "Add a new game"
    :script (ps ; client side validation
      (defvar add-form nil)
      (defun validate-game-name (evt)
        (when (= (@ add-form name value) ""))
          (chain evt (prevent-default))
          (alert "Please enter a name.")))
      (defun init ()
        (setf add-form (chain document
          (get-element-by-id "addform"))))
        (chain add-form
          (add-event-listener "submit"
            validate-game-name false)))
        (setf (chain window onload) init))))
    (:h1 "Add a new game to the chart")
    (:form :action "/game-added" :method "post" :id "addform"
      (:p "What is the name of the game?" (:br)
        (:input :type "text" :name "name" :class "txt"))
      (:p (:input :type "submit" :value "Add" :class "btn")))))

```

Towards robustness

I touched on the principles of unobtrusive JavaScript earlier. The term itself is not formally defined, but typically involves a separation of content (our HTML) from the behaviour (e.g. client-side validation). In the code above we're only half way there as we still include the validation code in our handler. This is a deliberate choice. In Lisp the separation between content and behaviour becomes less of an issue; I'm not actually writing JavaScript, everything is Lisp.

A more serious concern is cross-browser compatibility. The landscape of the web is notorious for the difficulty of making sure your code runs with all kinds of browsers our clients throw at our pages. For example, earlier versions of Internet Explorer expected us to use the function *attachEvent* rather than *addEventListener*. Writing code to support all these different versions is time-consuming and error prone. A much better alternative is to use a third-party library (for example [jQuery](http://jquery.com)¹⁷) that encapsulates all browser-specific quirks. Such a library will give you a

¹⁷<http://jquery.com>

more robust event handling mechanism that ensures even those poor souls stuck with vintage editions of proprietary browsers are able to benefit from our validation functions.

With Parescript it's straightforward to integrate any third-party JavaScript libraries. Better yet, we can add our own macros and abstractions on top of them to carve out the kind of language we'd like to have.

The Lisp advantage

The generated JavaScript code is simple enough that we could have written it directly. So why go through Lisp? I prefer it for the following reasons:

- Parescript gives us a uniform representation. We're now using the same syntax to express both HTML, JavaScript, and the actual application-specific logic residing on the server side.
- It lets us raise the level of abstraction to that of idiomatic Lisp. The big win with Parescript is that we get access to the macro system of Common Lisp. Using macros, we can be just as ruthless in eliminating repetitive code and recurring patterns on the client side as we were on the server side.
- Since Parescript constructs have access to most of the core Common Lisp language, our client-code will use familiar constructs. Most of the time, the code we define with Parescript will run just as fine in our REPL as it does in the browser.
- The last point is a bit more subtle, but I believe it to be an important one. Since we never have to leave our Lisp environment, we're able to avoid cognitive context switches. By staying in the same development context we're more likely to maintain our flow as we avoid the distractions of mentally switching to another syntax.

The final point is the reason I chose to include the script code directly in my handlers. As I spot duplications, I just extract and encapsulate the code as I would for any other Lisp functions. You could of course locate all client-side code in a module of its own. For rich clients, such an organisation is probably beneficial. With Parescript the choice is yours: it's Lisp all the way down.

Persistent Objects

Initially we kind of ducked the problem with persistence. To get things up and running as quickly as possible, we used a simple list in memory as “database”. That’s fine for prototyping but we still want to persist all added games in case we shutdown the server. Further, there are some potential threading issues with the current design. Hunchentoot is multi-threaded and page requests may come in different threads. We can solve all that by migrating to a real database. And with Lisp, design decisions like that are only a few forms away.

MongoDB as a backend

When it comes to persisting information, the software field is rich with alternatives. Since I wrote the original version of Lisp for the Web, NoSQL databases have gotten their fair share of hype. Choosing a solution requires us to understand the problem we’re trying to solve. When I’m not quite sure I try to delay that decision. My approach is to start out simple. Using Lisp and some discipline I know I can always change my decision later as I learn more through prototyping.

MonoDB is a database I often use in that early stage. Mongo happens to be a popular NoSQL database built around the idea of representing content as documents. The main reason I chose Mongo is that it’s simple to setup and comes with a programmer-friendly environment. In particular, I like its shell that lets me inspect, explore and play with the data.

From Lisp to Mongo and back again

After installing MongoDB, we need to start its *mongod* daemon process. We could have *mongod* run on its own node, but in early development stages I run *mongod* on localhost using the default port.

mongod will be our main point of interaction. As soon as we want to perform some operation on the backend or query some data we’ll send a message to *mongod*. The open-source library *cl-mongo* allows us to get up and running in now time. First, just load *cl-mongo* with Quicklisp:

```
RETRO-GAMES> (ql:quickload '(cl-mongo))
```

Once *cl-mongo* is loaded we can connect to the database *games* that we’ll use for all storage:

```
(cl-mongo:db.use "games")
```

MongoDB stores all documents in a collection. In our case we chose to introduce a *game* collection:

```
(defparameter *game-collection* "game")
```

We can now refer to the collection by using the symbol **game-collection** in our code.

Migrating to persistence

We took care earlier to encapsulate the access to the backend and now comes the time when we reap the benefits. We just have to change those functions to use the *cl-mongo* API instead of working with our **games** list:

```
(defun game-from-name (name)
  (let ((found-games (docs (db.find *game-collection*
                                   ($ "name" name)))))
    (when found-games
      (doc->game (first found-games)))))

(defun game-stored? (name)
  (game-from-name name))
```

The new implementation of *game-from-name* queries the **game-collection** for a game matching the given name. The function is built on *cl-mongo*'s *db.find*. By default *db.find* behaves like Mongo's *findOne* so when we *found-games* we know there can be only one. In that case, we invoke the function *doc->game* in order to restore our server side CLOS instance from the mongo document:

```
(defclass game ()
  ((name :reader name
         :initarg :name)
   (votes :accessor votes
          :initarg :votes ; when read from persistent storage
          :initform 0)))

(defun doc->game (game-doc)
  (make-instance 'game :name (get-element "name" game-doc)
                 :votes (get-element "votes" game-doc)))
```

The only new function here is *get-element*. It's provided by *cl-mongo* and allows us to extract the values of the fields in the retrieved document. The rest of the code passes the extracted values as *initargs* to *make-instance* that produces a CLOS object. Please note that I made a small extension to the *votes* slot. Since we want to read back its persisted value, I've extended it with an *:initarg* similar to how *name* works.

Just like our initial implementation using *find*, *game-from-name* returns NIL in case no object with the given name is stored. That means that we can keep *game-stored?* exactly as it is without any changes. But what about adding a new game? Well, we no longer need to maintain any references to the created objects. The database does that for us. But, we have to change *add-game* to insert a new document in the mongo database:

```
(defun add-game (name)
  (let ((game (make-instance 'game :name name)))
    (db.insert *game-collection* (game->doc game))))
```

let establishes a binding between the *game* symbol and a new instance of the *game* class. Using *db.insert* from *cl-mongo* we persist the document created by *game->doc* in the database. The extra step of instantiating a CLOS object isn't strictly necessary, but I'd like to keep related functions on the same level of abstraction. And that's the case with our new utility *game->doc*:

```
(defun game->doc (game)
  ($ ($ "name" (name game))
    ($ "votes" (votes game))))
```

Just like its counterpart in the other direction (*doc->game*), this function is responsible for the translation between the two domains. The syntax may look a bit funny at first - what's with all those dollar signs? Well, the symbol *\$* is a convenience macro provided by *cl-mongo*. The macro makes it easy to create a document and add the fields *name* and *votes* to it using a declarative programming style. Without the *\$* macro we'd had to use much more cumbersome constructs:

```
; painful, imperative style without the $ macro:
(defun game->doc (game)
  (let ((game-doc (make-document)))
    (add-element "name" (name game) game-doc)
    (add-element "votes" (votes game) game-doc)
    game-doc))
```

\$ may take some time to get used to. But as evident from comparing the samples above it does simplify the code.

Notes on concurrency

Our initial implementation based on the shared *games* list will run into trouble if accessed from multiple threads. Using MongoDB, we push the burden of at least this concurrency issue on the database.

Mongo resolves concurrent access using a multiple readers, single writer mechanism. Since a document in mongo is updated atomically, we won't get any race conditions in our code. Or will we? What if two users decide to add the same game simultaneously? Even if mongo does its job and ensures that the two insertions don't interfere with each other, we'll still end up with two documents representing the same game.

Avoid Duplicates with Constraints

To avoid duplicates in the database we first have to specify what we mean by a unique game. In the current version of the code, based on a limited domain model, the *name* alone is used for uniqueness. To have mongo enforce that domain rule, we have to specify a unique index constraint on our collection:

```
(defun unique-index-on (field)
  (db.ensure-index *game-collection*
    ($ field 1)
    :unique t))
```

I abstract the index creation in a function since I want to be clear with its purpose:

```
RETRO-GAMES> (unique-index-on "name")
```

Once we've run that code, we're safe for duplicates. Mongo will detect violations and enforce the constraint automatically.

On natural IDs

MongoDB ensures that each document we *db.insert* has a unique key. This is done through the `__id` field of each document. If we don't provide one, mongo will generate one for us. Now, it's tempting to use the *name* of each *game* as ID. Not only will it save us some space since the uniqueness of the `__id` field is already enforced. We'd also no longer need to provide our own constraints.

The problem with this approach is that we get more than we ask for. In many domains, choosing a natural ID is risky. For retro games, it turns out there are several cases where different games ended up with the same name. Similarly, it happened that games were released in one US version and one European version. Using the name as a key would get us into trouble. By letting mongo generate a unique ID we're free to evolve the domain model to include publisher, release year, etc.

As our model grows we may need to revisit our constraint. Perhaps it needs to grow into a compound based on multiple fields to express the true uniqueness of a game.

CLOS: Observers for free

Our original *vote-for* method is responsible for increasing the vote count for a given game:

```
(defmethod vote-for (game)
  (incf (votes game)))
```

On its own, *vote-for* no longer does its whole job. Obviously we need to propagate changed values to the mongo documents. One way would be to just add a call to *cl-mongo*'s *db.update* method directly in *vote-for*. However, that would compromise the integrity of the code. Suddenly, our *vote-for* method would have two responsibilities. To me, modifying an object and persisting possible updates are orthogonal functions better kept separate. Using CLOS, we can implement our new functionality and still keep the code clean and simple.

The object model of CLOS is radically different from mainstream object-orientation of today. Some problems that require elaborate class hierarchies in Java or C# are trivial in CLOS. One such example involves listening to method invocations. If we can get a notification each time *vote-for* is invoked, we could easily add our persistence mechanism to the code receiving the notification. And here CLOS provides an elegant solution by its concept of method combinations.

Like everything else in CLOS, method combinations are customizable. But often, the default *standard method combination* will do. The standard method combination works like this:

1. The programmer defines a primary method, perhaps specialised on a particular class. In our example, *vote-for* is a primary method.
2. Before the primary method is invoked, CLOS calls a possible *:before* method.
3. Symmetrically, once the primary method has run, CLOS invokes the *:after* methods, if any.
4. In addition, CLOS lets us specify *:around* methods. We won't use *:around* methods here, but basically they're run before any other method and specify when the next method is to be called. That next method may be any of the other ones: before, after, or primary - the *:around* method doesn't know, neither should it.

Once I got my head around the method combination model, I started to see possible applications everywhere. Suddenly, concepts that were previously hard to separate from the main body of code (for example logging and trace messages) can be encapsulated in methods of their own and kept entirely orthogonal to the main flow. Our need for persistence is precisely one such concept:

```
(defmethod vote-for :after (game)
  (let ((game-doc (game->doc game)))
    (db.update *game-collection* ($ "name" (name game)) game-doc)))
```

Since we want to persist an updated game, we specify an *:after* method. The *game* instance we receive has now been voted for and we can *db.update* the corresponding *game-collection* in mongo.

Common Lisp code doesn't tend to be particularly object-oriented (at least not in the traditional sense). Yet it's interesting to note that our usage of an *:after* method is actually a dynamic version of the design pattern Observer. Our sample also embodies the open-closed principle for real: we extended the behaviour of an existing class without modifying any code whatsoever. CLOS takes care of the object dispatch for us at just the right time.

Sorting games through MongoDB

Just one final change before we can deploy our new backend: the *games* function responsible for returning a list of all games sorted on popularity:

```
(defun games ()
  (mapcar #'doc->game
    (docs (iter
      (db.sort *game-collection*
        :all
        :field "votes"
        :asc nil))))))
```

Again we're relying on utilities in *cl-mongo*. *db.sort* is actually a macro that converts our code into a *find* query, tailored to work as a sort. We just specify that we want to retrieve *:all* games, have them sorted on the *:field* "votes", and invert the default sort order to get the games by descending vote rank. However, for efficiency reasons, mongo doesn't necessarily return the full set of matches. Instead we get a cursor back that we can iterate over. As an alternative, we can realise the full sequence (iterate to the end) using *iter* as we do above. To convert the resulting sequence of documents we re-use our previously defined *doc->game* function that we apply to each document using *mapcar*.

Remembering the Games

The *games* function completed our migration to a persistent storage. But we still want to keep all previously added games. After all, users shouldn't suffer because we decide to change the implementation. So, how do we transform existing games into persistent documents in MongoDB? The simplest way is to map over the **games** list, transform each *game* into a document, and insert the document in mongo:

```
RETRO-GAMES> (mapcar #'(lambda (old-game)
  (db.insert *game-collection* (game->doc old-game)))
  *games*)
```

We could have defined a function for this task using *defun* but, because it is a one-off operation, I go with an anonymous function aka *lambda* function. And that's it - all games have been moved into a persistent database. We can now set **games** to *NIL* (effectively making all old games available for garbage collection) and even make the **games** symbol history by removing it from the package:

```
RETRO-GAMES> (setf *games* nil)
NIL
RETRO-GAMES> (unintern '*games*)
T
```

MapReduce in Lisp

This tutorial illustrates my typical approach to software development; I start out simple, get a basic version working and evolve it iteratively learning as I go along. If we're careful with the additions we make, maintain a consistent style, and chose our libraries wisely, our code is more likely to support our future needs.

In this chapter, we'll throw one such new algorithm at our retro games to see how the design reacts. My choice of algorithm, MapReduce, is like the gateway drug to big data. Once you've understood its mechanism, a whole field of interesting applications opens up.

Pushing work to the server-side

Our Retro Games application is in a quite early stage. It has some limitations. It's also easy to picture future additions. One such possible feature is to allow users to categorize the added games. Such a feature would allow us to present charts by category such as action games, shoot 'em ups, adventures, etc. There's sure added user value behind such a feature. But I got more personal motivations - finally I will get an informed answer to whether [U.S.A.A.F.](http://www.lemon64.com/?mainurl=http%3A//www.lemon64.com/games/details.php%3FID%3D3386)¹⁸ really was a better strategy game than [Theatre Europe](http://en.wikipedia.org/wiki/Theatre_Europe)¹⁹.

Extending *:retro-games* to include a game category is straightforward. We start with our domain model:

```
(defclass game ()
  ((name      :reader    name
           :initarg  :name)
   (votes     :accessor  votes
           :initarg  :votes
           :initform 0)
   (category  :accessor  category
           :initarg  :category)))
```

The new slot *category* uses CLOS constructs that should be familiar by now. I've just added a new slot, *category*, that allows us to categorize each game instance. The necessary backend extensions also use familiar concepts:

¹⁸<http://www.lemon64.com/?mainurl=http%3A//www.lemon64.com/games/details.php%3FID%3D3386>

¹⁹http://en.wikipedia.org/wiki/Theatre_Europe

```
(defun game->doc (game)
  (with-slots (name votes category) game
    ($ ($ "name" name)
      ($ "votes" votes)
      ($ "category" category))))

(defun add-game (name category)
  (let ((game (make-instance 'game :name name
                              :category category)))
    (db.insert *game-collection* (game->doc game))))
```

Since we encapsulated the backend access in a thin layer of functions, our changes are local to that abstraction. As I added the category field to our game document, I also rewrote *game->doc* to use *with-slots* to get rid of some repetitive access of slots on the *game* instance. We've seen *with-slots* before as we customised the printing of CLOS objects. The macro simply allows us to access the slots as though they were normal variables. The function *add-game* is kept as-is.

The code above just added support for persisting the category slot. We know it will be a killer feature. But perhaps we want a simple way to monitor it, say by querying the number of games in each category? Such functionality would allow us to watch the individual charts grow as users flood to our app. We could model the functionality on top of our *games* function. Let's consider that option for a moment: *games* would have to retrieve all games, grouped by *category*, and then sum the number of games in each category. It's doable, but quite inefficient. Wouldn't it be better if we could somehow push the work to the database server? How would such a solution work?

Let's take a step back and reflect upon the game aggregation mechanism I just described. Basically, we want to iterate over all stored games and split the game collection into smaller sequences where each represent one category. Finally, we'd just sum-up the number of games in each sequence. That'd be the result: the number of games for each category. Possibly, we could parallelise the different stages in the pipeline. If you by now think that algorithm sounds familiar, you're absolutely right - it's [MapReduce](#)²⁰ of course!

The MapReduce algorithm in MongoDB

MongoDB opens up a lot of possibilities, all of them accessible to the Lisp REPL at your hands. Since MapReduce is such a fundamental algorithm for dealing with large volumes of data, MongoDB includes support for it by means of its `mapReduce` database command. All we have to do as a client is to supply the *map*- and *reduce*-functions to mongo. MongoDB will take care of them and execute the algorithm in its daemon process (mongod). Mongo's scripting language of choice is JavaScript. That means our *map*- and *reduce*-functions have to be defined as JavaScript too. That's excellent news to us - we've already seen how Parescript allows us to generate JavaScript from Lisp and now we can leverage that knowledge.

²⁰<http://en.wikipedia.org/wiki/Mapreduce>

Specifying the steps with Parescript

Let's start with our *map* function. According to the MongoDB documentation, the function will be applied to each document in our collection and is expected to return key-value pairs. For our purposes, we want to use *map* to separate out the *category* (our key) of each game document. Since we're interested in the total number of games in each category, we'd just return a constant 1 (one) as the value of each individual game:

```
(defjs map_category()
  (emit this.category 1))
```

defjs in the code above is a utility, provided by *cl-mongo*, that lets us define client-side JavaScript functions. The actual JavaScript code is expressed with Parescript. *defjs* just ensures the function is accessible in our upcoming mongo interactions. *emit* is a JavaScript generator that produces our key-value pair when invoked on a game document.

The reduce phase will collect the data produced by *map*. The *reduce* function will be invoked once for each aggregated sequence produced by the earlier step. Now *reduce* just needs to sum-up the votes:

```
(defjs sum_games(c vals)
  (return ((@ Array sum vals))))
```

We have already met the *\$* macro in the Parescript chapter. There we used the macro to access object properties. Since we want to invoke the function *sum* in the *Array* module, we surround the form with a pair of parentheses to turn it into a function call. Now the body of the function expands into the following JavaScript:

```
(function (c, vals)
{
  return Array.sum(vals);
});
```

With the two steps of MapReduce defined we're ready to go.

Executing MapReduce from the REPL

Like all multi-step high-level algorithms, MapReduce takes some experience to wrap ones head around. So let's recap what we have so far: *map_category* will be invoked on all documents and categorize each one of them based on the value of the *category* field. The MapReduce function then aggregates the emitted key-value pairs into groups. We'll get one group for each category. These sequences are now fed into the reduce phase which produces our results by summing up the number of games in each group. Everything runs on the server-side in our mongo daemon.

To glue it together, we just need to invoke the algorithm parameterized with our functions. Again *cl-mongo* comes to rescue by providing a helpful macro to interact with mongo's scripting engine. The *\$map-reduce* macro takes three mandatory arguments:

1. A collection to operate on. This corresponds to our collection of *game* documents.
2. A *map* function to apply. Here we'll use the *map_category* function we just introduced.
3. A *reduce* function, which correspond to *sum_games* defined above.

Just as I did earlier, I encapsulate the functionality in a *defun* to express the language of the domain:

```
(defun sum-by-category ()
  (pp (mr.p ($map-reduce "game" map_category sum_games))))
```

Since the REPL is the developers best friend, let's take our MapReduce implementation ²¹ for a test drive:

```
RETRO-GAMES> (add-game "Tetris" "Classic")
NIL
RETRO-GAMES> (add-game "Theatre Europe" "Strategy")
NIL
RETRO-GAMES> (sum-by-category)
{Classic
  "value" -> 1.0d0
}
{Strategy
  "value" -> 1.0d0
}
RETRO-GAMES> (add-game "U.S.A.A.F." "Strategy")
NIL
RETRO-GAMES> (sum-by-category)
{Classic
  "value" -> 1.0d0
}
{Strategy
  "value" -> 2.0d0
}
```

There are several things going on behind the smooth face of *sum-by-category*:

1. MapReduce in MongoDB just returns a result document specifying where to find the result, not the actual values produced by the algorithm. Instead, the resulting documents are stored in a result collection.
2. *mr.p* from *cl-mongo* hides that extra step by parsing the name of the result collection and then fetches the specified result document for us.
3. Finally, *pp* pretty prints the result in our REPL.

Executing MapReduce on MongoDB with JavaScript functions shows the main strength of Lisp; the language can morph into anything we want.

²¹I ran into what seems to be incompatibility problems with this implementation. If you get any problems, check out the *map_reduce_in_mongo.lisp* code in my [Github repository](#) for a workaround.

A natural extension: presenting charts by category

The previous extensions laid a foundation for our next step: assembling the charts. Again, an efficient way of working is to outsource as much of the work as possible to the server-side. Using *cl-mongo*'s *db.find* method with the appropriate selector allows us to tailor our *games* function to just match the documents of specific categories.

Writing the supporting HTML is straightforward and provides an excellent opportunity for me to leave as your guide. Our *standard-page* macro lets you put it together in no time. Now, will Theatre Europe make a final rise to the top? The stage is yours.

Endgame

Final considerations

This book has really just scratched the surface of what Lisp can do. Yet I hope that if you made it this far, you have seen that behind all those parenthesis there's a lot of power. With its macro system, Lisp can basically be what you want it to.

Due to the dynamic and interactive nature of Lisp it's a perfect fit for prototyping. And because Lisp programs are so easy to evolve, that prototype may end up as a full-blown product one day.

To guide you, Common Lisp provides several useful resources. In this final chapter I'll provide some advice and useful links.

On backends

SQL

Relational databases have been around for a long time. Unsurprisingly, there are several open-source libraries for interacting with common SQL databases. Just as we used CL-WHO to express HTML, and Parescript to generate JavaScript, the different SQL libraries often provide an embedded domain-specific language for writing SQL in Lisp. The level of abstraction obviously varies between libraries. I tend to stay as close to the SQL as possible since I found it easier to reason about the performance of my queries that way. But your mileage may vary.

Since there are many different alternatives and since your choice of driver will ultimately depend upon the backend you use, I recommend that you check out the database section of [The Common Lisp Wiki](#)²².

NoSQL alternatives

Today, many large scale applications chose schema-less databases. In this book we have already met MongoDB. But mongo is just one of several possible alternatives. If you're interested in this kind of backends, the following links lead to Common-Lisp open-source drivers for popular databases:

- [cloudbdb](#)²³ is a library for interacting with Apache CouchDB.
- [CL-Redis](#)²⁴ is a client library for interacting with the key-value store Redis.

²²<http://www.cliki.net/database>

²³<http://common-lisp.net/project/cloudbdb/>

²⁴<http://www.cliki.net/cl-redis>

- [Rucksack](#)²⁵ is a promising library that attempts to be a transparent persistence mechanism for Lisp objects. The library reminds me of Elephant that we'll meet in the following chapter.

Persistent objects protocols

In the original Lisp for the Web article I used [Elephant](#)²⁶ as my persistent backend. Elephant is a wickedly smart persistent object protocol and database. I was reluctant to give up on it, but unfortunately Elephant hasn't been maintained since 2009 and is no longer compatible with some popular Lisp environments of today.

Since Elephant provides such an elegant programming model, I still consider it worth to invest some time in it just for the learning experience and inspiration it provides. You can get an overview in [my original article](#)²⁷. Here I will just give a brief overview to give you an idea of how Elephant works.

To actually store things on disk, Elephant supports several back ends such as PostGres and SqlLite. In my example I used Berkeley DB, simply because it has the best performance with Elephant. To make objects persistent, Elephant provides a convenient *defpclass* macro that creates persistent classes. The *defpclass* usage looks very similar to Common Lisp's *defclass*, but it adds some new features; we'll use *:index* to specify that we want our slots to be retrievable by slot values:

```
(defpclass persistent-game ()
  ((name :reader name
        :initarg :name
        :index t)
   (votes :accessor votes
          :initarg :votes
          :initform 0
          :index t)))
```

The Elephant abstraction is really clean; persistent objects are created just like any other object:

```
RETRO-GAMES>(make-instance 'persistent-game :name "Winter Games")
#<PERSISTENT-GAME oid:100>
```

Once we have created an instance of a *persistent-game* all mutations on the object will be automatically persisted. That means, our old *vote-for* function will work out of the box with Elephant.

²⁵<http://common-lisp.net/projects/rucksack/>

²⁶<http://common-lisp.net/project/elephant/>

²⁷<http://www.adamtornhill.com/articles/lispweb.htm>

Moving on

A programmer approaching Common Lisp from an OO background is in for a challenge. I know, since I've made that transition myself. But it's a challenge well worth the effort; learning a high-level multi-paradigm language like Common Lisp will change the way you look at programming for the better. To ease the task, a Common Lisp newcomer is lucky to have several high-quality books available.

Book recommendations

Since its publication back in 2005, *Practical Common Lisp* by Peter Seibel has become a modern classic and the generally recommended introductory text on Common Lisp. I was lucky to read *Practical Common Lisp* as my first Lisp book. It's just great.

Once you've mastered the basics of the language and gained some experience I'd recommend *Paradigms of Artificial Intelligence Programming (PAIP)* by Peter Norvig. PAIP isn't really a book on artificial programming. At least not the kind of AI we've come to expect today. Given that the book is more than two decades old this is actually a good sign. It means that the field has progressed. Instead, the book's sub-title is more descriptive of its content: *Case studies in Common Lisp*. But it's as a general programming book that PAIP excels, particularly when it comes to coding style. PAIP is one of the best programming books available.

Learning a new programming paradigm is always a challenge. If you find that you struggle with the functional style frequently used in Lisp, *The Little Schemer* by Daniel P. Friedman and Matthias Felleisen is a book for you. The book isn't about a certain language or syntax. Rather it teaches you something immensely more valuable: how to think about programming. And it does it by building a Socratic dialog between the reader and the authors. The teaching style alone makes it worth a read.

Another solid introduction to a functional mindset is found in a different community. OCaml from the *Very Beginning* by John Whittington is a book I recommend to anyone interested in learning functional programming. OCaml is different from Lisp in both syntax and semantics, but you can carry the functional ideas from John's book to Lisp.

Finally, let's bring in the heavy artillery. You've consumed the books above and look for more challenging material. In that case, I'd suggest *On Lisp* by Paul Graham or *Let Over Lambda* by Doug Hoyte. Both books are well-written, engaging and of astonishing technical depth. If you want to know as much about macros as possible, these two books are your best companions on that journey.

You'll find reviews of these and many more high-quality Lisp books on my [book review pages](http://www.adamtornhill.com/bookreviews.htm)²⁸.

Code reading

Once I managed to master the basics of Lisp I learnt a lot by reading the code of other programmers. As we've seen in this book, Common Lisp is a low-ceremony language. If you

²⁸<http://www.adamtornhill.com/bookreviews.htm>

understand the problem domain it's usually quite easy to get a grip of the code too. As such, any of the open-source libraries we've used in this book is a good starting point. I personally enjoyed looking into *cl-mongo*'s source code. Another favourite is the code by [Edi Weitz](http://weitz.de)²⁹. Edi designed the Hunchentoot and CL-WHO packages that we used in this book. His source code is always well-documented and provides an excellent starting point for new Lisp programmers diving into the language.

Happy hacking! wishes

adam@adamtornhill.com

Sweden, January 2014

²⁹<http://weitz.de>