;;;;;;;;;;;;;;;;;;;;;;;;

;;;; Lisp Web Tales ;;;;

;;;;;;;;;;;;;;;;;;;;;;;;

;; My attempts at learning web development,
;; foolishly using Common Lisp,
;; and even more foolishly,
;; writing about it in public

;;;; A book By Pavel Penev

```lisp
(restas:define-module #:lisp-web-tales
    (:use #:cl #:restas))

(in-package #:lisp-web-tales)

(define-route about ("about/" :method :get)
  '(:features
    ("Simple tutorials using various lisp libraries"
     "An in-depth look into some of them"
     "My irrelevant oppinions on Lisp and the web"
     "Common Lisp cult propaganda for the unenlightened")
    :target-audience
    ("Anyone interested in Lisp"
     "People who find Node.js too mainstream")))
```

Lisp Web Tales

My attempts at learning web development, foolishly using common lisp, and even more foolishly, writing about it in public

Pavel Penev

This book is for sale at http://leanpub.com/lispwebtales

This version was published on 2013-11-24

Tweet This Book!

Please help Pavel Penev by spreading the word about this book on Twitter!

The suggested hashtag for this book is #lispwebtales.

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=#lispwebtales

Contents

Preface

I am an enthusiast if there was ever such a thing. So this is an enthusiasts book, written out of joy and curiosity, and as an escapist pleasure in a time when the outside world is closing in on me, and my time for lisp is running short. Exams, graduation, the eventual job search, and employment as a Blub coder is what is in front of me for 2013.

To me Lisp is one of the most fun and easy to use languages out there, it has challenged me intellectually, and provoked my thinking in all sorts of directions, from basic software design, to how software communities work. All of these questions have led me to believe that the right course for me personally is to continue to learn Common Lisp and its history. I will not be a worse programmer if I continue to invest effort into mastering it, just the opposite. The same is true for all sorts of languages and platforms, and some of them I am also investing my self in, such as GNU Emacs, Linux, and as horribly flawed as it is, the web. Whatever my day jobs might be in the future, I will continue my hobbyist practice as a programmer, and until I find a better tool, I will continue to use and love Common Lisp.

This book is in a way an attempt at maintaining that practice and getting my skill level up. It has taken a lot of research and experimentation, and helped me improve my writing. So even if it fails to attract an audience, and even if left unfinished, it is well worth the effort.

Pavel Penev, March 2013

Introduction

Why Lisp

Today we have more programming languages than we can count. Somehow, Lisp still manages to stand out, at least for me. I've been obsessed with the lisp family of languages for four years now, and I've been especially interested in Common Lisp, which I consider to be the best general purpose dialect. It is an easy language to pick up, and a difficult language to master. So far, every day spend learning lisp has been a huge plus for me, so all those difficulties have been worth it. Lisp is fun, challenging and rewarding of such efforts. No language I've picked up since or before has felt the same way, they were all either needlessly complex(most of the complexity in lisp is there for a reason), or too simplistic and lacking in sophistication when that is needed.

As for how practical this language is for web development, It's as practical as you make it. Lisp is the perfect language for the gray areas where were we still haven't quite figured out how to do things. I believe the web is one such area, and experimentation and playful exploration of ideas is vital. This is what Lisp was designed for, not for the web specifically, but for what it is, a new playground where flexibility and creativity have room to grow.

Common Lisp has been a faithful ally in my self-education. Maybe it can be one for you too.

Whats in the book

The book is a set of tutorials and examples. It uses the Common Lisp language and some of the libraries we'll be using for the examples and tutorials include:

- The Hunchentoot web server
- The Restas web framework
- The SEXML library for outputting XML and HTML
- Closure-template for HTML templating
- Postmodern for PostgreSQL access, and cl-reddis as a simple datastore
- Various utilities

Who is this for

This book is for anyone interested in Lisp and web apps. I assume you have some familiarity with both subjects, but I don't assume you are a Lisp expert, you can just read a few tutorials to get the basics and get back to my book to get started with web apps. I've linked some of them in Appendix B. So you need to know what `(+ a b)` means, I won't explain html and css to you, and HTTP shouldn't be a scary mystical acronym to you. Also some knowledge of databases would be good. In other words, I assume you are a programmer, know the basics and just want to play around with Lisp.

What you need to get started

A lisp implementation, preferably sbcl(recommended for Linux users) or ccl(recommended for Mac and Windows users), and Quicklisp, the Common Lisp package manager. I've written a quick "getting started" tutorial in Appendix A. And the links in Appendix B have additional information.

You will also need a text editor which supports matching parenthesis, so no notepad. Appendix A has some recommendations, but the best way to use Lisp is with Emacs and the Slime environment. A similar environment is available for Vim users with the Slimv plugin. If you don't already know Emacs or Vim, you can leave learning it for later, and just use any old code editor and the command line. If you are serious about Lisp though, consider picking up Emacs eventually.

Appendix B also has a lot of links you can go to to find more about Lisp, including tutorials, books, wikis and places you can ask your questions.

Typographic conventions

Inline code:

This code is inlined: `(lambda () (format t "Hello World"))`.

This is a code block in a file:

```
(defun hello-world ()
  (format t "Hello World"))
```

The following characters represent various prompts:

A * represents a lisp REPL, => marks the returned result:

```
 * (format nil "Hello World")
 => "Hello World"
```

$ is a unix shell, # is a root shell, or code executed with `sudo`:

```
 # apt-get install foo
 $ foo --bar baz
```

> is a windows `cmd.exe` prompt:

```
 > dir C:\
```

1 The basics

Raw example

Here is a complete hello-world web application, saved in the file `hello-world.lisp`:

```lisp
;;;; hello-world.lisp

(ql:quickload "restas")

(restas:define-module #:hello-world
    (:use :cl :restas))

(in-package #:hello-world)

(define-route hello-world ("")
  "Hello World")

(start '#:hello-world :port 8080)
```

This apps basically returns a page with the text "hello world" to any request at the "/" uri. It can be run from the command line using sbcl or ccl like this:

```
$ sbcl --load hello-world.lisp
```

or

```
$ ccl --load hello-world.lisp
```

Or loaded from the lisp prompt:

```
 * (load "hello-world.lisp")
```

Now you can open the page http://localhost:8080/[1] and see the result.

Detailed explanation

I'll do an almost line by line explanation of what is happening.

[1]http://localhost:8080/

```
(ql:quickload "restas")
```

All examples in this book will be using the hunchentoot web server, and the RESTAS web framework built on top of it.

As you can read in the Appendix A, the way we install and load libraries with Quicklisp is with the `quickload` function. The `ql:` part simply means that the function is in the ql package, which is a short name for the `quicklisp` package. Lisp packages often have such short alternative names, called nicknames. This line simply loads Restas, and installs it if it isn't already present. Since hunchentoot is a dependency for Restas, it gets loaded as well.

```
(restas:define-module #:hello-world
    (:use :cl :restas))

(in-package #:hello-world)
```

Restas applications live in modules, which are similar to ordinary common lisp packages(and in fact, a package is being generated behind the scenes for us), we define them with the macro `define-module` from the `restas` package. It has the same syntax as common lisps `defpackage`. We give our module the name `hello-world` and specify that we want all public symbols in the `cl` and `restas` packages to be imported into our module. We then set the current package to `hello-world`. All the code after this form to the end of the file will be in that package.

Symbols starting with `#:` are uninterned, meaning they have no package, we just want to use its namestring, which is `"HELLO-WORLD"`. Uninterned symbols are useful if you want a lightweight string to name something, in this case a package.

The following form `(:use :cl :restas)` means that all the "public" symbols from the packages `cl`(a standard package containing all lisp functions, variables, classes etc) and `restas` get imported into our `hello-world` package, so we don't have to write `restas:define-route` and can simply say `define-route`.

```
(define-route hello-world ("")
  "Hello World")
```

Restas apps are based on uri handlers called routes. Routes in their simplest form shown here, have: * A name (`hello-world` in this case) * An uri template. in this case the empty string `""`, meaning it will match the `/` uri * A body generating a response, in this case the string "hello world" returned to the client.

There are a few more details to routes, but we'll get to them in a bit.

```
(start '#:hello-world :port 8080)
```

The Restas function `start` is used to initializes a module, and starts a hunchentoot web server. As a first argument we give it the symbol naming our module with our application defined in it and pass a port number as a keyword parameter. Note that the symbol must be quoted ith a `'`. Again, there is quite a bit more to this function, but for now, we just need to get our app running.

A simple blog

Lets look at a bit more complicated example: a simple blog app. It will be self contained in a single file you can run from the command line, just like the previous example. Subsequent examples will use ASDF and Quicklisp. In addition to Restas and Hunchentoot we'll also be using the SEXML library for html generation. The blog posts will be stored in memory as a list. The basic features would be:

- View all blog posts on the front page
- Separate pages for each post
- Separate pages for authors, listing all of their posts.
- Admin form for adding posts, protected by crude HTTP authorization.

The source code

Here is the complete source of our app, consisting of slightly over 100 lines of code:

```
;;;; blogdemo.lisp

;;;; Initialization

(ql:quickload '("restas" "sexml"))

(restas:define-module #:blogdemo
  (:use #:cl #:restas))

(in-package #:blogdemo)

(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))

(<:augment-with-doctype "html" "")

(defparameter *posts* nil)

;;;; utility

(defun slug (string)
  (substitute #\- #\Space
              (string-downcase
               (string-trim '(#\Space #\Tab #\Newline) string))))

```

```
;;;; HTML templates

(defun html-frame (title body)
  (<:html
   (<:head (<:title title))
   (<:body
    (<:a :href (genurl 'home) (<:h1 title))
    body)))

(defun render-post (post)
  (list (<:div
         (<:h2 (<:a
                :href (genurl 'post :id (position post *posts* :test #'equal))
                (getf post :title)))
         (<:h3 (<:a
                :href (genurl 'author :id (getf post :author-id))
                "By " (getf post :author)))
         (<:p (getf post :content)))
        (<:hr)))

(defun render-posts (posts)
  (mapcar #'render-post posts))

(defun blogpage (&optional (posts *posts*))
  (html-frame
   "Restas Blogdemo"
   (<:div
    (<:a :href (genurl 'add) "Add a blog post")
    (<:hr)
    (render-posts posts))))

(defun add-post-form ()
  (html-frame
   "Restas Blogdemo"
   (<:form :action (genurl 'add/post) :method "post"
    "Author name:" (<:br)
    (<:input :type "text" :name "author")(<:br)
    "Title:" (<:br)
    (<:input :type "text" :name "title") (<:br)
    "Content:" (<:br)
    (<:textarea :name "content" :rows 15 :cols 80) (<:br)
    (<:input :type "submit" :value "Submit"))))

;;;; Routes definition

```

```
(define-route home ("")
  (blogpage))

(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (blogpage (list post))))

(define-route author ("author/:id")
  (let ((posts (loop for post in *posts*
                  if (equal id (getf post :author-id))
                  collect post)))
    (blogpage posts)))

(define-route add ("add")
  (multiple-value-bind (username password) (hunchentoot:authorization)
    (if (and (equalp username "user")
             (equalp password "pass"))
        (add-post-form)
        (hunchentoot:require-authorization))))

(define-route add/post ("add" :method :post)
  (let ((author (hunchentoot:post-parameter "author"))
        (title (hunchentoot:post-parameter "title"))
        (content (hunchentoot:post-parameter "content")))
    (push (list :author author
                :author-id (slug author)
                :title title
                :content content) *posts*)
    (redirect 'home)))

;;;; start

(start '#:blogdemo :port 8080)
```

This file can be run from the command line like so:

```
$ sbcl --load blogdemo.lisp
```

or

```
$ ccl --load blogdemo.lisp
```

Or load it from the Lisp prompt:

```
 * (load "blogdemo.lisp")
```

The username and password for adding new posts, as can be seen in the source, are “user” and “pass” respectively. Try adding posts, and vary the names of authors. Explore how the app behaves. In later chapters we will learn how to improve it a bit, but for now, it will do.

Source walk-through

Lets walk through the various sections of this source code and see how it works.

Initialization

```
(ql:quickload '("restas" "sexml"))
```

We begin by loading the libraries we’ll be using: Restas and sexml.

```
(restas:define-module #:blogdemo
  (:use #:cl #:restas))

(in-package #:blogdemo)
```

This time our application is named `blogdemo`.

```
(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))

(<:augment-with-doctype "html" "")
```

SEXML is a library for outputting XML using lisp s-expressions as input. It takes an xml dtd and generates a package with functions for all the necessary tags. In our case, we give it an html5 dtd, and specify the package named `<`. This means that we can write code like:

```
(<:p "Hello world")
```

and get this out:

```
<p>Hello world</p>
```

A thing to note is that SEXML comes with an html5 dtd file as part of the distribution. The code `(merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))` is used to find the path to that file. Don’t worry about how this actually works, essentially it means “give me the path to the file ‘html5.dtd’ in the sexml installation directory”.

And finally, we define our “database” as an empty list named by the variable `*posts*`:

```
(defparameter *posts* nil)
```

Utility

I've included a section for utility functions, which at this point contains only one function:

```
(defun slug (string)
  (substitute #\- #\Space
              (string-downcase
               (string-trim '(#\Space #\Tab #\Newline)
                            string))))
```

If you are familiar with Django, you probably know the term 'slug'. A slug is a string we can use in urls. The `slug` function takes a string, such as `" Foo Bar BaZ "` and converts it to a url friendly string like `"foo-bar-baz"` by trimming surrounding white space, converting all the characters to lower case and substituting the spaces between words for dashes. We'll be using it to create ID's for authors in our "database".

HTML templates

In general the rules for using sexml for html generation are as follows:

```
(<:tagname attributes* content*)
```

where attributes can be of the form:

```
:key value
```

and the content can be a string or a list of strings to be inserted into the body of the tag. For example, this snippet:

```
(<:a :href "/foo/bar" "This is a link to /foo/bar")
```

Will produce the following HTML: `<a href="/foo/bar">This is a link to /foo/bar</a>`

Lets take a look at the various template functions we'll be using:

```
;;;; HTML templates

(defun html-frame (title body)
  (<:html
   (<:head (<:title title))
   (<:body
    (<:a :href (genurl 'home) (<:h1 title))
    body)))
```

`html-frame` is a function, which takes a title and a body and converts it to an html page whose body has a link to the home page at the top. We can call it like so:

```
(html-frame "This is a title" "This is a body")
```

And get the following output as a lisp string(I've indented it, and broken it up to separate lines):

```
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <a href="/"><h1>This is a title</h1></a>
    This is a body
  </body>
</html>
```

Of note is the use of the restas function `genurl` which takes a route, and generates a url that would be handled by the route function. In this case `(genurl 'home)` will generate the `/` url, since that is what the `home` route(defined in the next section) handles. This is done because restas applications can me "mounted" on different points in a url path tree. if the whole application is mounted on the uri `/blog`, then the same code(without having to change it) would generate `/blog/` as the output.

Before we look at how we generate the blog posts themselves, let me explain how we store them. We store blog posts as a list of plists, a convention for lists where the odd numbered elements are keys, and the even numbered elements are values. Plists are useful as lightweight maps, and look like this:

```
'(:author "Author Name"
  :author "author-name" ; this is a slug string
  :title "This is a title"
  :content "This is the body of the blog post")
```

By convention, keys in plists are common lisp keywords, starting with colons. Elements in a plist can be accessed with the function `getf`, which takes a plist and a key as it's argument. So if we wanted to get the name of the author from the plist `post`, we would write `(getf post :author)`. Simple as that. Now lets look at how we use them:

```
(defun render-post (post)
  (list (<:div
         (<:h2 (<:a
                :href (genurl 'post :id (position post *posts* :test #'equal))
                (getf post :title)))
         (<:h3 (<:a
                :href (genurl 'author :id (getf post :author-id))
                "By " (getf post :author)))
         (<:p (getf post :content)))
        (<:hr)))

(defun render-posts (posts)
  (mapcar #'render-post posts))
```

The function `render-post` takes a blog post and renders it as html. The `genurl`'s in this function are a bit more complicated. In this case `genurl` has to generate urls to each individual post, which requires additional information, such as it's ID. We use the posts position is the list of posts as it's id, so each post would have a url like `posts/1` or whatever it's number in the list is. Same is true for the author, except authors are identified by a slug of their name. so the url would look like `author/author-name`. This works because routes can handle more than one url with similar structure, for instance both `posts/1` and `posts/2` will be handled by the route `post`, We'll see how that works in a minute.

The function `render-posts` simply takes a list of posts, and renders each one individually, into a list of html strings. It uses the `mapcar` function, which might be called `map` or `each` in other languages.

```
(defun blogpage (&optional (posts *posts*))
  (html-frame
   "Restas Blogdemo"
   (<:div
    (<:a :href (genurl 'add) "Add a blog post")
    (<:hr)
    (render-posts posts))))
```

`blogpage` takes a bunch of blog posts and renders a complete html page with them. By default it renders all of the posts, but we can give it a subset, as we do when we show only the posts by one author.

And finally, `add-post-form` generates a page with an html form in it for adding a blog post:

```
(defun add-post-form ()
  (html-frame
   "Restas Blogdemo"
   (<:form :action (genurl 'add/post) :method "post"
    "Author name:" (<:br)
    (<:input :type "text" :name "author")(<:br)
    "Title:" (<:br)
    (<:input :type "text" :name "title") (<:br)
    "Content:" (<:br)
    (<:textarea :name "content" :rows 15 :cols 80) (<:br)
    (<:input :type "submit" :value "Submit"))))
```

This is it for html generation.

Routes

Route handlers are the heart of any Restas application. The complete syntax for `define-route` is:

```
(define-route name (template &key method content-type)
  declarations*
  body*)
```

We've seen a very basic usage of this. The blog example doesn't use the optional declarations, we'll cover them later, but the optional method keyword parameter will come in handy when we need to handle `POST` data from a form. By default it's value is `:get`, but can be any HTTP method. Using this we can have routes with the same uri template, but different HTTP methods, this makes Restas very well suited for RESTful APIs as the name suggests. Let's take a look at the individual routes:

```
(define-route home ("")
  (blogpage))
```

Simply displays the home page.

```
(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (blogpage (list post))))
```

The `post` route handles the case where we are viewing a single blog post. We see that the `post` route has an interesting template: `"post/:id"`. If you are familiar with something like Sinatra, you'll find this syntax familiar. Parts of a uri template beginning with a colon designate route variables, and can match any uri with that structure, as I mentioned in the previous section. For example `/post/0`, `/post/1` and `/post/2` will all match and be handled by this route. But so will `/post/foo`, our app breaks if we go to a url that doesn't have an integer url component. We'll see later how we can fix this, for now, we simply don't do that. The matched string also gets bound to a lisp variable in the body of the route, in our case `id`.

Lets look at the body, each such route template variable is bound to the string that it matched, so we get values like `"0"`, or `"1"` or `"foo"`. Using common lisps `parse-integen` we convert it to an integer, and we look up the element in the list of posts with the `elt` function, which takes a list(or any lisp sequence) and gets the item at the index we suply as a second argument.

We render the post by passing it as a list to `blogpage`, which returns a string, which in turn, Restas returns to the browser.

```
(define-route author ("author/:id")
  (let ((posts (loop for post in *posts*
                  if (equal id (getf post :author-id))
                  collect post)))
    (blogpage posts)))
```

The `author` route is very similar. we have an `:id` variable as well, but it can be any string, so we don't worry about parsing it. We use common lisps powerful loop macro to iterate over all the posts, and if the id we supply in the url matches the `:author-ids` of the individual posts, we collect them into a new list. `:author-id` is generated as a slug version of the author name, specifically so that we can use it as a key and as a part of a url.

If we have blog posts by an author named `"Pavel Penev"`, its slug would have been saved it into the database as `"pavel-penev"`, and if we go to the uri `author/pavel-penev`, we'll see all the posts by that author on the page.

```
(define-route add ("add")
  (multiple-value-bind (username password) (hunchentoot:authorization)
    (if (and (equalp username "user")
             (equalp password "pass"))
        (add-post-form)
        (hunchentoot:require-authorization))))
```

The `add` route handles displaying a form for the user to submit a blog post. Since we don't want just anybody to add posts, we want to add some user authentication, but since this is just a simple example, I won't bother with login forms, cookies and sessions, we'll leave that for a later chapter. For now I'll use simple HTTP authorization.

If you are unfamiliar with HTTP authentication, it is a very crude way to log into a web site. The browser has to supply a username and a password as an HTTP header. The function `hunchentoot:authorization` returns them as two separate values, since common lisp supports multiple return values, instead of just one(as is the case in probably every other language you've ever used), we have to bind them using the macro `multiple-value-bind`, which is like `let` for multiple values. It binds the variables `username` and `password` and in the body we check if they are the expected values, in our case "user" and "pass". If they are, we render our form, and if not, we tell the browser to ask the user for a username and password using 'hunchentoot:require-authorization'.

```
(define-route add/post ("add" :method :post)
  (let ((author (hunchentoot:post-parameter "author"))
        (title (hunchentoot:post-parameter "title"))
        (content (hunchentoot:post-parameter "content")))
    (push (list :author author
                :author-id (slug author)
                :title title
                :content content) *posts*)
    (redirect 'home)))
```

Finally, `add/post` handles the data send to us by the form generated by `add`. We specify that the http method should be `POST`, and use the function `hunchentoot:post-parameter` to extract the user submitted values from the request. The strings `"author"`, `"title"` and `"content"` were the names of fields in our form. We bind them to values, and built a plist using the function `list`. Note that we add the `:author-id` key, with a value generated by applying `slug` to the `author` string. The list we `push` onto the database variable `*posts*`. `push` takes an item, and adds it to the front of a list. At the end we `redirect` back to the home page. `redirect` is a restas function with much the same syntax as `genurl` but instead of generating a url out of a route name, it generates the url, and tells the browser to redirect to it.

Conclusion

This concludes the honeymoon chapter. We saw all the very basic ideas: A restas web application is a module, which is a collection of route functions that handle uri requests. There is quite a bit more to it than that, and we'll get to it in the next chapters. The code was kind of bare bones, usually we would like to have an ASDF system defined, so we can have all the lisp infrastructure available for us (have Quicklisp download all of our dependencies, have our templates be compiled automatically, and be able to develop interactively). At the moment our apps are bare scripts, and lisp is not a scripting language, even though you can use it as such. It's a powerful interactive system, and we would be fools if we didn't take advantage of that power.

2 Setting up a project

Single file apps are fine and dandy, but things will start getting ugly as we increase the complexity of our applications. In this chapter we'll learn about ASDF systems and how to structure our projects. I'll also introduce a small utility for generating a restas project skeleton.

Systems

ASDF is a build system for Common Lisp. An ASDF system contains information on the files that are part of your app, and in what order are they to be compiled and loaded into the system. It also knows about the dependencies of your application. Lisp systems come with ASDF, and if you installed Quicklisp, you also have an easy way to download and load existing ASDF systems.

Defining an ASDF system is easy, the definition is contained in a small file in your project directory with an `.asd` extension. We'll see in a moment what such a file looks like. Code in the ASD file is lisp, so you shouldn't have problems reading and understanding it.

Quicklisp and manual installation

We already saw how libraries can be downloaded, installed and loaded with Quicklisp, but what if a library isn't in Quicklisps repository? When you installed QL, it created a directory called `quicklisp` in your home directory. This is where Quicklisp installs all of the libraries it downloads. Inside it is another directory called `local-projects`. This directory contains locally installed libraries, that weren't downloaded with Quicklisp. Every ASDF system in `local-projects` is instantly visible to quicklisp, and it can compile and load that system. That is mighty convenient. That is where we'll be putting all of our projects so that we can load the source.

restas-project

I have written a small utility called `restas-project` for generating a project skeleton for Restas applications. It is modeled after another utility called `quickproject` which is more general purpose and simply generates a Common Lisp application skeleton. We could use that, but `restas-project` is more convenient for our needs, since we'll only be making Restas apps.

Since it is not yet in Quicklisp, we have to install it manually. You can get it at Github[1]. You can either download an archive and uncompress it in `local-projects`, or if you know how to use git, simply clone the repository into `quicklisp/local-projects/` to make it available:

[1]https://github.com/pvlpenev/restas-project

```
$ cd ~/quicklisp/local-projects
$ git clone git@github.com:pvlpenev/restas-project.git
```

Setting up a hello-world project

First start lisp:

```
$ sbcl
```

Then load restas-project:

```
 * (ql:quickload "restas-project")
```

After it finishes loading lets create our `hello-world` project:

```
 * (restas-project:start-restas-project "hello-world")
```

This will create a new directory called `hello-world` in `local-projects`. Lets see what we have in there:

```
.
├── defmodule.lisp
├── hello-world.asd
├── hello-world.lisp
├── static/
│   ├── css/
│   ├── images/
│   └── js/
├── templates/
└── tests/
```

`hello-world` contains 3 files and 3 directories. `static` is for any static content we might have, it's 3 sub-directories `css`, `images` and `js` are for css, image files and JavaScript source files respectively. The `templates` directory is for template files we'll write later on. And the `tests` directory is for putting unit tests. Since we didn't tell `restas-project` to generate a test ASDF system, this directory will be unused for now.

hello-world.asd

Let's look at the files. First we have the system definition file `hello-world.asd`. Let's have a peak inside:

```lisp
(defpackage #:hello-world-config (:export #:*base-directory*))
(defparameter hello-world-config:*base-directory*
  (make-pathname :name nil :type nil :defaults *load-truename*))

(asdf:defsystem #:hello-world
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS)
  :components ((:file "defmodule")
               (:file "hello-world")))
```

The first 3 lines define a config package, we use it simply to have a reference in our running application to the directory the source is located, since we'll need access to static resources that aren't lisp source files, like css files and template files. These lines are not that important, the important part is the `defsystem` definition, let's look it line by line.

The first argument to `defsystem` is the system name, in this case `#:hello-world`. The next line `:serial t` tells asdf that we want our lisp files to be loaded in the order we specify them, since code in latter files might depend on previous files.

Following are 3 description lines. The keys `:description`, `:author` and `:license` specify additional info about our system, the system will still run fine without them, but it is a good idea to include that information here. For now, we won't bother with editing it.

Next, with the key `:depends-on` we list our library dependencies. In this case it contains only Restas, but can have any number of names of asdf systems.

The `:components` key tells ASDF what files are in our system. In this case only `defmodule.lisp` and `hello-world.lisp`. Note the syntax, components are specified as a list of lists, starting with the keyword `:file` followed by the name of the file, without the extension. So `defmodule.lisp` is specified as `(:file "defmodule")`.

defmodule.lisp

In the file `defmodule.lisp` we define our packages and modules included in our application. Normal Lisp projects call this file `packages.lisp` by convention, but we use the Restas convention of naming it `defmodule.lisp`. Lets look inside:

```
;;;; defmodule.lisp

(restas:define-module #:hello-world
  (:use #:cl))

(in-package #:hello-world)

(defparameter *template-directory*
  (merge-pathnames #P"templates/" hello-world-config:*base-directory*))

(defparameter *static-directory*
  (merge-pathnames #P"static/" hello-world-config:*base-directory*))
```

The first lisp form should seem mostly familiar, it simply defines a module named `#:hello-world` like we did in the previous chapter. Note however that the script did not add `#:restas` as a package to be imported into the `hello-world` package. This is because I personally prefer to qualify symbol names with their package name, so I would write `restas:define-route` instead of `define-route`. If you prefer otherwise, change that form to `(:use #:cl #:restas)`.

Next we define two variables, called `*template-directory*` and `*static-directory*`, if we happen to need to find the files that might be present there.

This is it for `defmodule.lisp`.

hello-world.lisp

This is the main file of our application. This is where all of our routes will be defined. Here is how this file should look like:

```
;;;; hello-world.lisp

(in-package #:hello-world)

;;; "hello-world" goes here. Hacks and glory await!
```

Nothing but an ‘in-package‘ clause. Let’s not keep it empty, let’s define a route:

```
(restas:define-route main ("")
  "Hello World")
```

Running the project

Now let’s start our project. First open up lisp

```
$ sbcl
```

Then load `hello-world` with Quicklisp:

```
(ql:quickload "hello-world")
```

Then start up the server:

```
(restas:start '#:hello-world :port 8080)
```

Now if you navigate to http://localhost:8080[2] you'll see the message `Hello World`

You can stop the server either by exiting lisp, or by typing the following:

```
(restas:stop-all)
```

Conclusion

That's it for project organization for now. Next up, we'll explore HTML generation and output with templates.

[2] http://localhost:8080

3 HTML generation and templating

Exploring all the options

HTML generators

An html generator is a library to generate html from lisp code. SEXML, the library we saw earlier is an example of one such generator. There are as many such libraries as there are lisp programmers, I'll lie to you if I told you I didn't try writing one myself. Cliki[1] lists 12, and that is certainly not all of them. The most popular such library is `cl-who` and it is the most likely for you to come across if you read other peoples code. But I, and many others don't like `cl-who`(not that it's bad, just not my taste), so I just picked another one for the examples in this book. Namely SEXML.

A note about SEXML, when we want to use it in a project, we'll add it's initialization code to the `defmodule.lisp` file:

```
(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))

(<:augment-with-doctype "html" "")
```

In chapter one, since you didn't yet know what asdf was, I didn't explain the line `(asdf:system-source-directory "sexml")`. ASDF here finds us the directory where Quicklisp installed SEXML.

`(<:augment-with-doctype "html" "")` simply means that if we use the `<:html` function, the result will have an html5 doctype line added to the result.

Template languages

In most other languages, you don't have dsl's for outputting html, like in lisp, so you tend to use templates. Common Lisp has a few of those. Most popular is probably `html-template`[2] by Hunchentoot creator(among other things, including cl-who) Edi Weitz. Its documentation page claims it is loosely based on a similar Perl library.

The one I'm going to be using for some of my examples is the Common Lisp port of Google's Closure Template system, written by the creator of Restas, Andrey Moskvitin.

[1]http://www.cliki.net/HTML%20generator
[2]http://weitz.de/html-template/

closure-template

For more, read the excellent [documentation][docs] and check out the examples at the github repo[3].

Concepts

A closure template can be defined either in a file or a string. Both starting with a namespace declaration, and subsequent template definitions. Here is a simple example:

```
{namespace hello}

{template main}
  <h1>Hello World</h1>
{/template}
```

Once compiled, this code will generate a Common Lisp package named `hello` and containing the function `main`. Let's try it at the REPL:

```
 * (ql:quickload "closure-template")
 * (defparameter *template* "{namespace hello}
{template main}
  <h1>Hello World</h1>
{/template}")
 * (closure-template:compile-template :common-lisp-backend *template*)
 * (hello:main)
 => "<h1>Hello World</h1>"
```

Each template namespace would usually live in a file ending in `.tmpl` by convention, and be compiled in our files.

The template commands have a fairly simple syntax: The command name, enclosed in `{}`.

Notice that the `{template}` tag had to be closed with `{/template}`, some tags like namespace don't need to be closed, while others do.

(Note: There can be only one namespace declaration per file.)

If we saved our template in a file, named say `main.tmpl`, we can compile it and run it at the repl like so:

```
 * (ql:quickload "closure-template")
 * (closure-template:compile-cl-templates #P"/path/to/main.tmpl")
 * (hello:main)
 => "<h1>Hello World</h1>"
```

I'll explain the rest of the syntax we'll need as we go along.

[3] https://github.com/archimag/cl-closure-template

More complex example

Lets create a new Restas project where we can play around with closure templates. Let's call it `closure-hello`:

```
 * (ql:quickload "restas-project")
 * (restas-project:start-restas-project "closure-hello" :depends-on '(:closure-template))
```

Create the file templates/main.tmpl and put the following code into it:

```
{namespace closure-hello.view}

{template main}
  <html>
    <head>
      <title>{$title}</title>
    </head>
    <body>
      {$body | noAutoescape}
    </body>
  </html>
{/template}
```

Now let's tell ASDF that we want this file to be compiled when we load our project, we do this by adding two things to the `defsystem` form in `closure-hello.asd`. First is the option `:defsystem-depends-on (#:closure-template)`, because ASDF needs closure-template in order to know how to compile closure templates. Second, we must specify the file as a component of type `:closure-template`, so our `defsystem` should look like this:

```
(asdf:defsystem #:closure-hello
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :defsystem-depends-on (#:closure-template)
  :depends-on (:RESTAS :CLOSURE-TEMPLATE)
  :components ((:closure-template "templates/main")
               (:file "defmodule")
               (:file "closure-hello")))
```

Now, let's load the system, and see if our template compiled:

```
 * (ql:quickload "closure-hello")
 * (closure-hello.view:main)
 => "<html> <head> <title></title> </head> <body>  </body> </html>"
```

Let's examine the new syntax in the template. Tags starting with a `$`, like `{$title}` simply print the content of the variable `title` to be into the output of the template. It's actually a short hand for the expression `{print title}`, but since it is used very often it was shortened to just `$`. Variables are passed to templates like as plists. Like so:

```
 * (closure-hello.view:main '(:title "hello" :body "<h1>world</h1>"))
 => "<html> <head> <title>hello</title> </head> <body> <h1>world</h1> </body> </html>"
```

And here is the output pretty printed, just so you can see that it worked:

```
<html>
  <head>
    <title>hello</title>
  </head>
  <body>
    <h1>world</h1>
  </body>
</html>
```

The | in the expression `{$body | noAutoescape}` is used to give the `print` command optional directives. In this case the `noAutoescape` directive, which turns off auto escaping, which is on by default and would escape the html we want to put into the body. This would make the browser not render it, but display the html directly. For example if the directive wasn't there, this is what we would get:

```
 * (closure-hello.view:main '(:title "hello" :body "<h1>world</h1>"))
 => "<html> <head> <title>hello</title> </head> <body> &lt;h1&gt;world&lt;/h1&gt; </body> \
</html>"
```

The `<h1>` tag got replaced with `&lt;h1&gt;`, and `</h1>` with `&lt;/h1&gt;`.

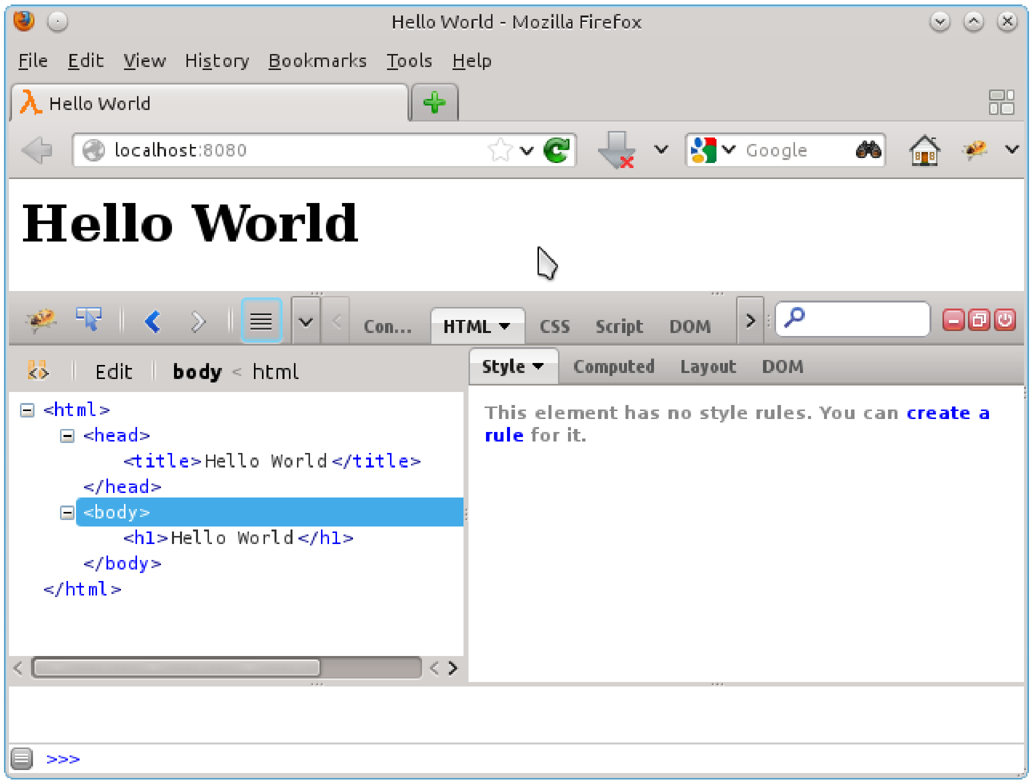
Lets define a handler, this is how the closure-hello.lisp file looks like:

```
;;;; closure-hello.lisp

(in-package #:closure-hello)

;;; "closure-hello" goes here. Hacks and glory await!

(restas:define-route main ("")
  (closure-hello.view:main
   (list :title "Hello World"
         :body "<h1>Hello World</h1>")))
```

Now we just quickload it, and start the server:

```
 * (ql:quickload "closure-hello")
 * (restas:start '#:closure-hello :port 8080)
```

Confirm that it works in the browser:


Hello World - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Hello World
localhost:8080
Google
Hello World
Con... HTML CSS Script DOM
Edit body < html
Style Computed Layout DOM
>>>

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

This element has no style rules. You can create a rule for it.

Yay!

Next step: Lets do some loops.

The data we pass to the templates can be either plists like we already saw, or they can be just lists of items we can handle with loops. For example `(list "val1" "val2" "val3")`. And of course we can nest them as we wish. We'll define a new page, where we will list a bunch of items in an html list. Lets say we have a list of TODO items, here is how the template would look like, add this code to the main.tmpl file:

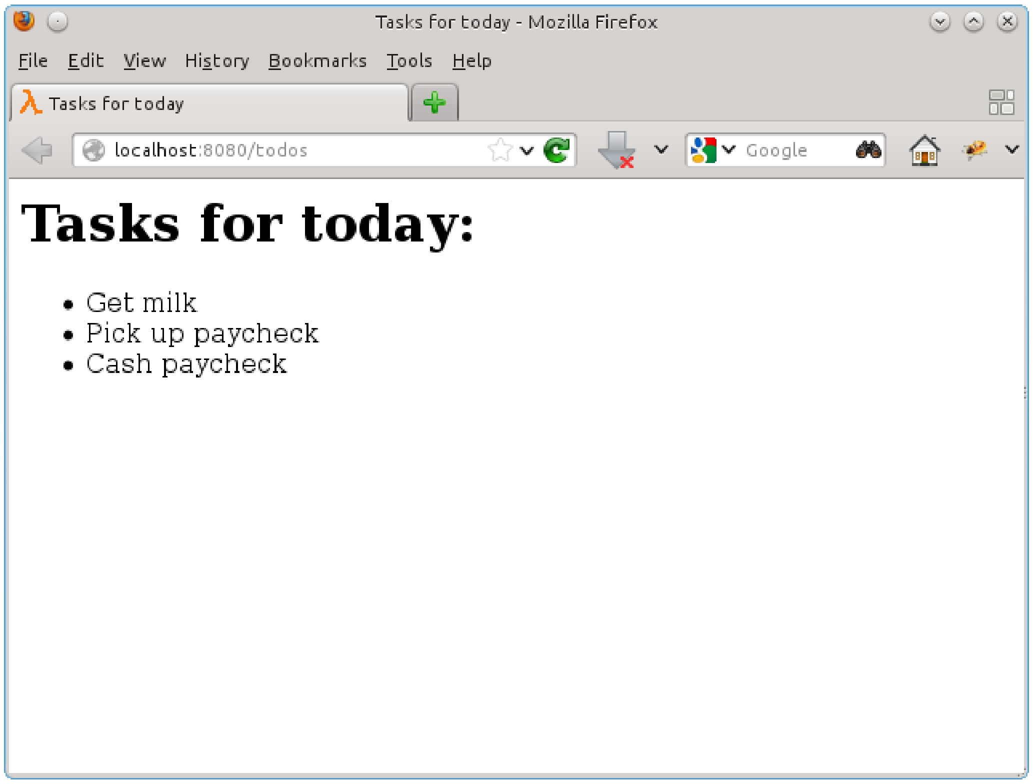
```
{template todos}
  <h1>Tasks for today:</h1>
  <ul>
    {foreach $task in $todos}
      <li>{$task}</li>
    {/foreach}
  </ul>
{/template}
```

`$todos` is the variable we pass directly to the template, foreach will loop through the contents, and put each item in the variable `$task`, which we can use in the body. The body outputs each list item.

Now lets add the handler in `closure-hello.lisp`, first we define a variable named *todos*, and then we define a restas route to handle requests to todos/, here is how the whole file should look like:

```
;;;; closure-hello.lisp

(in-package #:closure-hello)

;;; "closure-hello" goes here. Hacks and glory await!

(defparameter *todos* (list "Get milk" "Pick up paycheck" "Cash paycheck"))

(restas:define-route main ("")
  (closure-hello.view:main
   (list :title "Hello World"
         :body "<h1>Hello World</h1>")))

(restas:define-route todos ("todos")
  (closure-hello.view:main
   (list :title "Tasks for today"
         :body (closure-hello.view:todos (list :todos *todos*)))))
```

Notice that the code for the `todos` route is pretty much the same as for `main`, but we pass the output of the `todos` template to the body parameter of the `main` template, here is how it should look like in the browser when you navigate to the [http://localhost:8080/todos] url:


Tasks for today - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Tasks for today
localhost:8080/todos
Google

Tasks for today:

- Get milk
- Pick up paycheck
- Cash paycheck

Adding logic

Closure has support for conditionals, we'll only take a look at a simple example using `if`. The syntax is pretty straight-forward:

```
{if <expression1>}
  ...
{elseif <expression2>}
  ...
{else}
  ...
{/if}
```

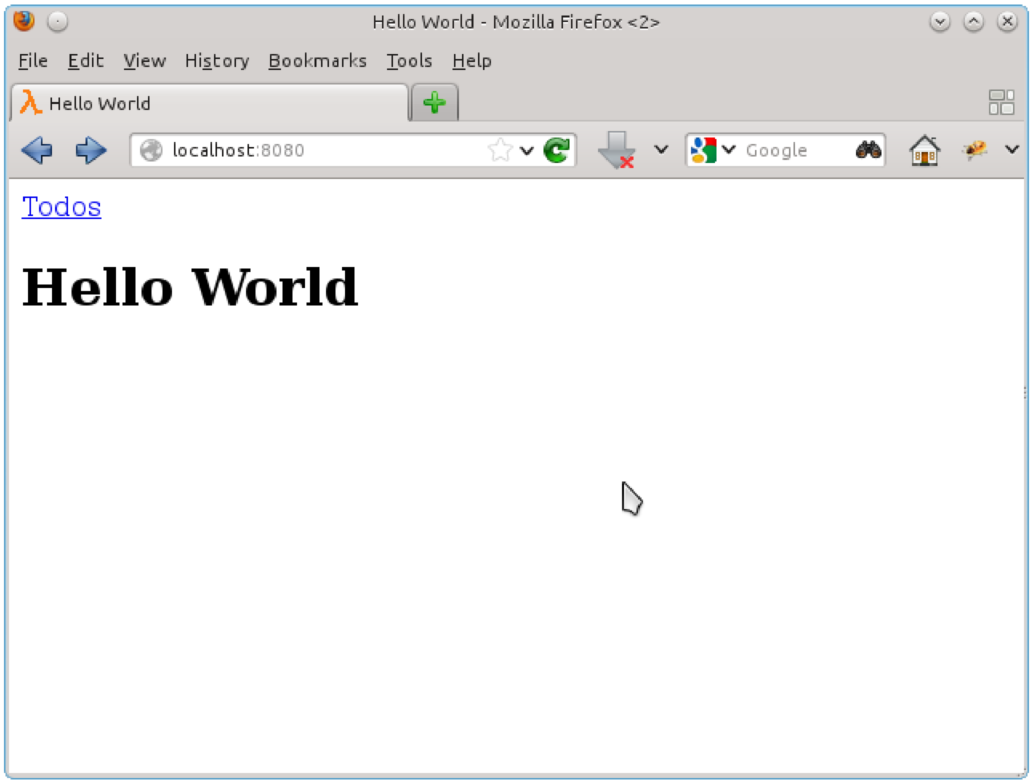
We want to add a link at the top of our main page to our todos page, but we also want to add a link to the main page, on our todos page. We'll add a conditional in our `main` template to check on which page we are, and put the appropriate link there. Here is how our template should look like:

```
{template main}
  <html>
    <head>
      <title>{$title}</title>
    </head>
    <body>
      {if $main}
        <a href="/todos">Todos</a>
      {elseif $todos}
        <a href="/">Home</a>
      {else}
        <h1>Where am i?</h1>
      {/if}
      {$body | noAutoescape}
    </body>
  </html>
{/template}
```

There are two new variables that we could pass to the main template, `$main`, which if true, means we are on the main page, and need to link to `todos/`, and `$todos`, which means we need to link to the home page. If none are passed, we output a message: “Where am i?”. We need to modify our routes, to pass the new parameters and tell the template what to do, we will also add a `lost/` page that doesn’t have any of the parameters:

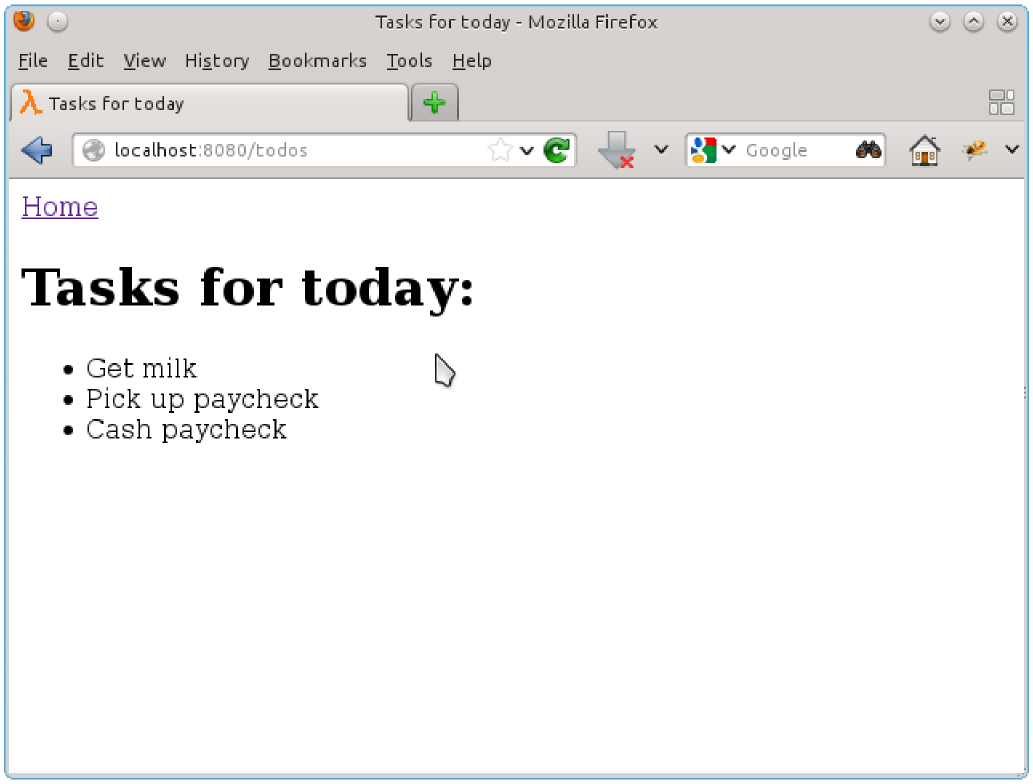
```
;;;; closure-hello.lisp

(in-package #:closure-hello)

;;; "closure-hello" goes here. Hacks and glory await!

(defparameter *todos* (list "Get milk" "Pick up paycheck" "Cash paycheck"))

(restas:define-route main ("")
  (closure-hello.view:main
   (list :title "Hello World"
         :main t
         :body "<h1>Hello World</h1>")))

(restas:define-route todos ("todos")
  (closure-hello.view:main
   (list :title "Tasks for today"
         :todos t
         :body (closure-hello.view:todos (list :todos *todos*)))))

(restas:define-route lost ("lost")
  (closure-hello.view:main
   (list :title "Are we lost?")))
```

And here are the screenshots:


Hello World - Mozilla Firefox <2>
File Edit View History Bookmarks Tools Help
Hello World
localhost:8080
Google

Todos

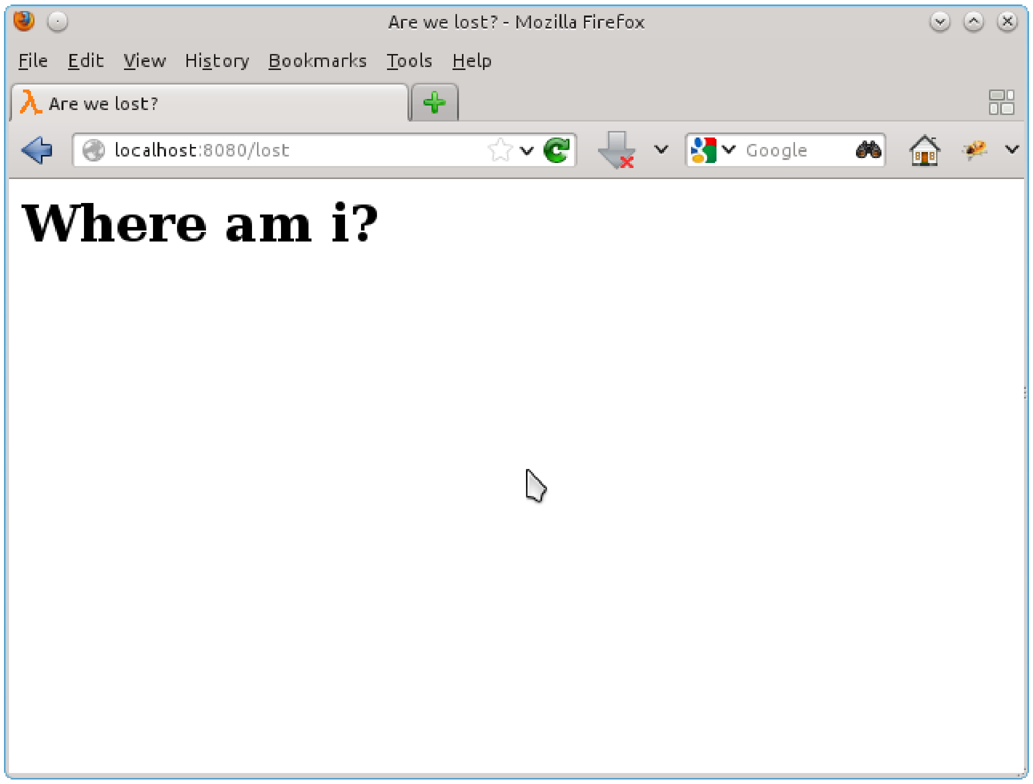
Hello World


Tasks for today - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Tasks for today
localhost:8080/todos
Google

Home

Tasks for today:

- Get milk
- Pick up paycheck
- Cash paycheck


Are we lost? - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Are we lost?
localhost:8080/lost
Google

Where am i?

About routes and templates.

A common pattern we see is that all of the routes is that they all call the same template, `closure-hello.view:main` to render their content. It would be nice to tell Restas to use the same template on all of the routes. I haven't talked in depth about how routes work, but in general, a route can return several types of values, if it returns an integer, say `404`, that will be interpreted as an HTTP status code, and an appropriate page will be returned to the user. If it returns a string, that string will be the body of the response, the routes we've seen so far are this kind. A route can also return a common-lisp pathname object, in which case hunchentoot will serve that file to the user. We can have a route like this in order to serve a static css file:

```
(define-route css ("main.css" :content-type "text/css")
  #P"/path/to/css/main.css")
```

And finally, a route can return a list, in which case something called the routes default render method will be called on the list, and the result returned to the user. There are two ways to specify a default render method, one is in the route definition, using a declaration. For example, we could have written 'main' this way:

```
(restas:define-route main ("")
  (:render-method #'closure-hello.view:main)
  (list :title "Hello World"
        :main t
        :body "<h1>Hello World</h1>"))
```

But we'd have to do that for all of the routes, and they all use the same render method. We could instead have a default render method for the entire module, in `defmodule.lisp`:

```
(restas:define-module #:closure-hello
  (:use #:cl)
  (:render-method #'closure-hello.view:main))
```

And then have all of the routes simply return lists:

```
;;;; closure-hello.lisp

(in-package #:closure-hello)

;;; "closure-hello" goes here. Hacks and glory await!

(defparameter *todos* (list "Get milk" "Pick up paycheck" "Cash paycheck"))

(restas:define-route main ("")
  (list :title "Hello World"
        :main t
        :body "<h1>Hello World</h1>"))
```

```

(restas:define-route todos ("todos")
  (list :title "Tasks for today"
        :todos t
        :body (closure-hello.view:todos (list :todos *todos*))))

(restas:define-route lost ("lost")
  (list :title "Are we lost?"))
```

Pretty neat, huh?

Tips for Emacs users.

closure-template-html-mode is in Marmalade, so you can install it from there. You can also use the chord `C-c C-l` to compile template files using slime, if `closure-template` is loaded into your lisp image.

Conclusion

That's about all the closure syntax you need for now, I encourage you to go through all of the documentation:

Google's documentation[4]

Complete list of closure commands[5], most of them are supported by the lisp version.

The examples in the github repo[6]

Documentation in Russian[7]

Keep in mind that in order to keep the examples simpler I will not use `closure-template` heavily for the rest of this book. In fact there is a lot I left out in this tutorial, since I will probably not need it. I aim to keep this short and simple.

[4]https://developers.google.com/closure/templates/
[5]https://developers.google.com/closure/templates/docs/commands
[6]https://github.com/archimag/cl-closure-template
[7]http://archimag.lisper.ru/docs/cl-closure-template/index.html

4 Putting it together: Updated blog example

Let's update our blog example, we'll turn it into an ASDF system, and update the templates slightly. Also, we'll add a session based authentication, with a login form to replace the crude http based authorization.

Creating the project

First, let's create the project:

```
 * (ql:quickload :restas-project)
 * (restas-project:start-restas-project "blogdemo" :depends-on '(:sexml))
```

Let's edit the `blogdemo.asd` to add a template.lisp file for our SEXML templates:

```
(asdf:defsystem #:blogdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML)
  :components ((:file "defmodule")
               (:file "template")
               (:file "blogdemo")))
```

Setting up defmodule.lisp

We'll need to edit `defmodule.lisp` to add `restas` in the `:use` list of `define-module` form:

```
(restas:define-module #:blogdemo
  (:use #:cl #:restas))
```

Following is the auto-generated code from `restas-project`:

```
(in-package #:blogdemo)

(defparameter *template-directory*
  (merge-pathnames #P"templates/" blogdemo-config:*base-directory*))

(defparameter *static-directory*
  (merge-pathnames #P"static/" blogdemo-config:*base-directory*))
```

Next we define the `*posts*` variable:

```
(defparameter *posts* nil)
```

and finally we paste the `sexml` initialization code:

```
(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))
```

Updating the templates

First, Let's create a template.lisp file, like we specified in the `asd` file. We set it's package to be `blogdemo` and specify that sexml should augment it's output with a doctype:

```
;;;; template.lisp

(in-package #:blogdemo)

(<:augment-with-doctype "html" "" :auto-emit-p t)
```

I've gone a bit crazy with redefining `html-frame`. When we start our app with `restas:start` we'll specify `html-frame` as the default render method for the module. Since by convention render methods take a plist of data to be displayed as a single parameter, that's what the new version will take. That parameter we'll call `context`. I've also defined a menu with a link to the home page, and conditionally displaying either a link to add a post or log out, or a link to log in.

```
(defun html-frame (context)
  (<:html
   (<:head (<:title (getf context :title)))
   (<:body
    (<:div
     (<:h1 (getf context :title))
     (<:a :href (genurl 'home) "Home") " | "
     (if (hunchentoot:session-value :username)
         (list
           (<:a :href (genurl 'add) "Add a blog post") " | "
           (<:a :href (genurl 'logout)
                (format nil "Logout ~A"
                        (hunchentoot:session-value :username))))
         (<:a :href (genurl 'login) "Log in"))
     (<:hr))
    (getf context :body))))
```

Since We'll be using the hunchentoot session to store information about the logged in user, we check if the session value `:username` has a value. If it is nil, there is no logged in user, and we display a link to the `login` route we'll define later. If the value is non-nil though, the user can add a post and can also log out, so we display the appropriate links, and display the username of the logged in user.

The `render-post` function remains the same:

```
(defun render-post (post)
  (list
   (<:div
    (<:h2 (<:a :href
               (genurl 'post :id (position post *posts* :test #'equal))
               (getf post :title)))
    (<:h3 (<:a :href
               (genurl 'author :id (getf post :author-id))
               "By " (getf post :author)))
    (<:p (getf post :content)))
   (<:hr)))
```

The other old template we need will be `add-post-form`, it is almost the same, except we don't wrap the code in `html-frame` here:

```
(defun add-post-form ()
  (<:form :action (genurl 'add/post) :method "post"
      "Author name:" (<:br)
      (<:input :type "text" :name "author")(<:br)
      "Title:" (<:br)
      (<:input :type "text" :name "title") (<:br)
      "Content:" (<:br)
      (<:textarea :name "content" :rows 15 :cols 80) (<:br)
      (<:input :type "submit" :value "Submit")))
```

And now for the login form:

```
(defun login-form ()
  (<:form :action (genurl 'login/post) :method "post"
      "User name:" (<:br)
      (<:input :type "text" :name "username")(<:br)
      "Password:" (<:br)
      (<:input :type "password" :name "password") (<:br)
      (<:input :type "submit" :value "Log in")))
```

These are all the template functions we need.

Rewriting the routes

The `home` route is slightly more complicated than before:

```
(define-route home ("")
  (list :title "Blogdemo"
        :body (mapcar #'render-post *posts*)))
```

The old `post` route looked like this:

```
(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (blogpage (list post))))
```

Updating it wouldn't be too difficult:

```
(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (list :title (getf post :title)
          :body (render-post post))))
```

But what if we call the URL `/post/blah`, that would break our program. We need to make sure that `id` is an integer, and signal a `404 not found` to the user otherwise. We already saw that we can add a declaration to a route to specify a render method, but we can also specify other things in declarations. One such thing is the `:sift-variables` declaration, which is used to validate and transform route variables. Lets use it here with `#'parse-integer`:

```
(define-route post ("post/:id")
  (:sift-variables (id #'parse-integer))
  (let ((post (elt *posts* id)))
    (list :title (getf post :title)
          :body (render-post post))))
```

This works, but we now have another problem to solve, what if the user enters a link to a post that doesn't exist yet, if we have only 3 posts, and the user enters `/post/33` the program will break again. Lets define a custom validation function to use with `:sift-variables`:

```
(defun validate-post-id (id)
  (let ((id (parse-integer id :junk-allowed t)))
    (if (< id (length *posts*))
        id
        nil)))
```

And now for the final version of the route:

```
(define-route post ("post/:id")
  (:sift-variables (id #'validate-post-id))
  (let ((post (elt *posts* id)))
    (list :title (getf post :title)
          :body (render-post post))))
```

For the `author` route we'll also want to add such validation. First we'll need to generate a list of all authors in the database, the following function collects all the `author-ids` in the database:

```
(defun get-authors ()
  (loop for post in *posts*
       collect (getf post :author-id)))
```

Now the validation function its self:

```
(defun validate-author-id (author-id)
  (find author-id (get-authors) :test #'string=))
```

And the route:

```
(define-route author ("author/:id")
  (:sift-variables (id #'validate-author-id))
  (let ((posts (loop for post in *posts*
                 if (string= id (getf post :author-id))
                 collect post)))
    (list :title (format nil "Posts by ~a" (getf (first posts) :author))
          :body  (mapcar #'render-post posts))))
```

Next, let's handle logging in and out. The `login` route is pretty simple:

```
(define-route login ("login")
  (list :title "Log in"
        :body (login-form)))
```

We'll handle the form in the `login/post` route. Again, we'll just check for the username and password being "user" and "pass" respectively. If they match, we start a hunchentoot session, set the session value of `:username` to "user" and redirect back to the home page. If the login wasn't successful, we redirect back to the login page:

```
(define-route login/post ("login" :method :post)
  (if (and (string= "user" (hunchentoot:post-parameter "username"))
           (string= "pass" (hunchentoot:post-parameter "password")))
      (progn
        (hunchentoot:start-session)
        (setf (hunchentoot:session-value :username) "user")
        (redirect 'home))
      (redirect 'login)))
```

Logging out is simply setting the session value to nil, and redirecting to the home page:

```
(define-route logout ("logout")
  (setf (hunchentoot:session-value :username) nil)
  (redirect 'home))
```

Now lets handle adding blog posts. Another declaration we can use in routes is `:requirements`. You supply it a predicate, and if it returns nil, restas returns a NOT FOUND page. We'll define such a predicate, since we only want our route to be accessible to logged in users:

```
(defun logged-on-p ()
  (hunchentoot:session-value :username))
```

And the route its self looks like this:

```
(define-route add ("add")
  (:requirement #'logged-on-p)
  (list :title "Add a blog post"
        :body (add-post-form)))
```

The `add/post` route that handles the `add` form is unchanged, and so is the slug function:

```
(defun slug (string)
  (substitute #\- #\Space
    (string-downcase
      (string-trim '(#\Space #\Tab #\Newline)
                   string))))

(define-route add/post ("add" :method :post)
  (let ((author (hunchentoot:post-parameter "author"))
    (title (hunchentoot:post-parameter "title"))
    (content (hunchentoot:post-parameter "content")))
    (push (list :author author
                :author-id (slug author)
                :title title
                :content content)
          *posts*)
    (redirect 'home)))
```

Post chapter refactoring

The app is still not pretty enough. Let's reorganize it a bit. The functions `slug`, `validate-post-id`, `get-authors`, `validate-author-id`, and `logged-on-p` are utility functions, let's move them to a file called util.lisp. That way `blogdemo.lisp` can contain only routes. First, let's add the file to `blogdemo.asd`:

```
(asdf:defsystem #:blogdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML)
  :components ((:file "defmodule")
               (:file "util")
               (:file "template")
               (:file "blogdemo")))
```

Note that `util.lisp` comes before `template.lisp` and `blogdemo.lisp`, since I'll want to use `logged-on-p` in `html-template` in a little bit. Another function I want to put in `util.lisp` is `start-blogdemo` to start our restas app:

```
(defun start-blogdemo (&optional (port 8080))
  (start '#:blogdemo :port port :render-method 'html-frame))
```

Note that it is here that we specify the render method for the module. We couldn't have done it in the `define-module` form, since the symbol `blogdemo::html-template` didn't exist before the package was created, so we would have to define a separate package for the templates, like `closure-template` does and export all of the symbols from that package so we can use them in the routes. On top of that, since we use route symbol names in the templates to generate urls, we would have to either export all of the route names from `blogdemo` or rewrite the templates not to use `genurl`. Before a fix included in a restas update to allow for a `:render-method` argument to `restas:start` I considered doing exactly this. Fortunately I didn't have to, since it would have made the code a lot more complicated.

Next, we need to export `start-blogdemo` from the `blogdemo` module, in `defmodule.lisp`:

```
(restas:define-module #:blogdemo
  (:use #:cl #:restas)
  (:export #:start-blogdemo))
```

Now we can go back and "fix" `html-frame` to use `logged-on-p`:

```
(defun html-frame (context)
  (<:html
   (<:head (<:title (getf context :title)))
   (<:body
    (<:div
     (<:h1 (getf context :title))
     (<:a :href (genurl 'home) "Home") " | "
     (if (logged-on-p)
         (list (<:a :href (genurl 'add) "Add a blog post")
           " | "
           (<:a :href (genurl 'logout)
             (format nil "Logout ~A"
                    (hunchentoot:session-value :username))))
         (<:a :href (genurl 'login) "Log in"))
     (<:hr))
    (getf context :body))))
```

Running the app

We are ready to start our app now:

```
 * (ql:quickload "blogdemo")
 * (blogdemo:start-blogdemo)
```

And we can stop the app of course with `(restas:stop-all)`.

Congratulations! We have come a long way since chapter 1, but we still have a lot of work to do!

5 Persistence part I: PostgreSQL

Introduction

I think we've written enough horrible code for the sake of simplicity and now I want to focus on "doing it right", to the best of my abilities, meaning, I want to use a real database. The example app I'll show you in the next few chapters will be a simple Reddit-like link sharing site. As an exercise, you can later go back and redo the previous blog example app to use a database.

In this chapter I'll use PostgreSQL and the excellent `postmodern` library, which is specific to Postgres. Other options exist, from the popular "NoSQL" data stores like Redis and MongoDB, to lisp specific object stores, like `bknr-datastore`. We'll implement a database layer in several of these stores. I'll show you how to use the Restas policy mechanism to define a single interface to the various back-ends.

Setting up PostgreSQL

Linux

Install postgresql using your distros package manager, for Debian based distros this looks like:

```
# apt-get install postgresql
```

I also usually install the graphical admin tool `pgadmin3` but it is optional.

Next we need to set up a database and a user for it. Postgres automatically creates a system user account named `postgres` for administration of the database server. Log in from the shell, and start `psql`(the postgres shell) so we can configure it to our needs:

```
# su - postgres
$ psql
```

Next, we need to create a pg user and a database, I'll name the user `linkdemouser` and the database `linkdemo`, then we quit with the `\q` command:

```
postgres=# CREATE USER linkdemouser WITH PASSWORD 'mypass';
postgres=# CREATE DATABASE linkdemo OWNER linkdemouser;
postgres=# \q
```

Log out as the user `postgres`, and were done:

```
$ exit
```

Windows

You can download a PostgreSQL graphical installer for windows from the PostgreSQL site[1]. Installation is straightforward, and you should probably follow the default installation options.

On windows 7 you will have to run the installer as an Administrator. The installer will ask you for a password for the super user `postgres`, you will use this DB account only to create a new user and a database.

At the end of the installation, deselect the check-box "Launch Stack Builder at exit?", we won't be needing that. Click finish.

After the installation completes, click the start menu, navigate to the PostgreSQL sub-menu and open "SQL Shell(psql)". It will prompt you for a server name, database, port and user. Press Enter on all four for the default values. Then it will ask for the `postgres` user password, type it in and press Enter.

Next, we need to create a pg user and a database, I'll name the user `linkdemouser` and the database `linkdemo`, then we quit with the `\q` command:

```
postgres=# CREATE USER linkdemouser WITH PASSWORD 'mypass';
postgres=# CREATE DATABASE linkdemo OWNER linkdemouser;
postgres=# \q
```

That's it.

What is a policy?

Although we don't need to do so, we'll use the restas policy mechanism to define an interface to our database. But first lets have a short discussion of the problem policies solve for us.

In our app, if we need to access the database we'll have a bunch of queries. Usually they are encapsulated in functions. So for example we might have a set of function like `(find-user id)` and `(auth-user username password)` which both contain queries written using postmodern. Lets say we want to have the option of using MySQL or some other backend and be able to switch between them. Common Lisp has a powerful OO system that allows us to do this easily. In our project we can define a variable called for example `*datastore*` and depending on its value a different implementation of our database layer gets used. Using generic functions this is easy, we simply define a class for each layer, for example:

```
(defclass postmodern-datastore () ...)
(defclass mysql-datastore () ...)
```

And set the `*datastore*` variable to an instance of one such class:

[1] http://www.postgresql.org/download/windows/

```
(setf *datastore* (make-instance 'postmodern-datastore ...))
```

Now, the functions `find-user` and `auth-user` can be defined in terms of generic functions:

```
(defun find-user (id)
  (datastore-find-user *datastore* id))

(defun auth-user (username password)
  (datastore-auth-user *datastore* username password))
```

Here `datastore-find-user` and `datastore-auth-user` are both methods defined on the `postmodern-datastore` and `mysql-datastore` classes, an instance of which we pass as the first argument. This pattern is fairly common, and restas provides a mechanism, called a policy, for generating all of the boilerplate necessary for it, such as defining the generic functions, the dispatch variable, the functions that call the methods, and optionally, various packages to put all of the stuff in, etc.

Creating the project

Now let's get on with writing our link sharing site. Let's create the project skeleton with `restas-project`:

```
 * (ql:quickload "restas-project")
 * (restas-project:start-restas-project
    "linkdemo"
    :depends-on '(:sexml
                  :postmodern
                  :ironclad
                  :babel))
```

Other than postmodern and sexml, we'll need the `ironclad` and `babel` libraries which will be used to hash the user passwords.

Defining the policy

Our next order of business is to define our policy, and all of the packages we'll be using. `restas:define-policy` has the following syntax:

```
(restas:define-policy <policy-name>
  (:interface-package <interface-package-name>)
  (:interface-method-template <interface-method-template>)
  (:internal-package <internal-package-name>)
  (:internal-function-template <internal-function-template>)

  (define-method <method-name> (args...)
    "Documentation string")

  <more method-definitions>)
```

What this does is:

- Defines a dynamic variable with the same name as the policy: `*policy-name*`
- Defines an interface package where all of the generic functions will be defined. This package will be used by the policy implementation we'll write for the specific backends
- Defines an internal package where the functions calling the methods will reside. This is the package our app will be using to access the database, instead of using the generic functions directly
- We can specify name templates for the functions and the methods. For example if we define a policy method `foo` we can say that we want the corresponding generic function to be called `generic-foo` and the internal function to be called `foo-bar`. We do this by specifying a format string(the new names are generated with `format`) to the declarations in the policy definition: `"GENERIC-~A"` and `"~A-BAR"` respectively.

In our case, this is the concrete example declaration we'll be putting in `defmodule.lisp`, also containing two method declarations for `find-user` and `auth-user`(there are more methods, but we'll add them later):

```
(restas:define-policy datastore
  (:interface-package #:linkdemo.policy.datastore)
  (:interface-method-template "DATASTORE-~A")
  (:internal-package #:linkdemo.datastore)

  (define-method find-user (username)
    "Find the user by username")

  (define-method auth-user (username password)
    "Check if a user exists and has the suplied password"))
```

The policy is named `datastore`, which means that the dynamic variable controlling dispatch will be named `*datastore*`. This variable is defined in the internal package, in our case named `linkdemo.datastore`. This package will also include the functions we actually call in our app, such as `find-user`. The interface package is called `linkdemo.policy.datastore` and this is where the generic functions that define our interface to the database are defined.

Notice the declaration of `:interface-method-template`. The declaration means that we want the generic functions in the interface package to be renamed according to the template `"DATASTORE-~A"` so for instance the generic function for `find-user` will be named `datastore-find-user`. I opted to skip defining such a rule for the functions in the internal package, but I could have done the same thing using `:internal-function-template`.

Also notice that method declarations are done with `define-method`. Do not be confused! Methods in Common Lisp are defined with `defmethod`, and here `define-method` is just part of the syntax of the `define-policy` macro. The argument lists of these method declarations will be the same as the functions in `linkdemo.datastore`. The argument lists of the generic functions in `linkdemo.policy.datastore` will have an extra argument called `datastore` which will be used for dispatch. For example `(find-user username) -> (datastore-find-user datastore username)`.

Here is the complete interface we will define today, complete with all the methods we need. Put this at the top of `defmodule.lisp`:

```
(restas:define-policy datastore
  (:interface-package #:linkdemo.policy.datastore)
  (:interface-method-template "DATASTORE-~A")
  (:internal-package #:linkdemo.datastore)

  (define-method init ()
    "initiate the datastore")

  (define-method find-user (username)
    "find the user by username")

  (define-method auth-user (username password)
    "Check if a user exists and has the suplied password")

  (define-method register-user (username password)
    "Register a new user")

  (define-method upvoted-p (link-id username)
    "Check if a user has upvoted a link")

  (define-method upvote (link-id user)
    "upvote a link")

  (define-method post-link (url title user)
    "post a new link")

  (define-method get-all-links (&optional user)
    "Get all of the links in the datastore")

  (define-method upvote-count (link-id)
    "get the number of upvotes for a given link"))
```

Defining the rest of the packages

Next, we need to define the restas module for our application, and the package where we will implement the policy interface for PostgreSQL, put this code after the policy declaration in `defmodule.lisp`:

```
(restas:define-module #:linkdemo
  (:use #:cl #:restas #:linkdemo.datastore))

(defpackage #:linkdemo.pg-datastore
  (:use #:cl #:postmodern #:linkdemo.policy.datastore))

(in-package #:linkdemo)

(defparameter *template-directory*
  (merge-pathnames #P"templates/" linkdemo-config:*base-directory*))

(defparameter *static-directory*
  (merge-pathnames #P"static/" linkdemo-config:*base-directory*))
```

Notice that linkdemo "uses" the internal package `linkdemo.datastore` where all of the functions like `find-user` are defined, and `linkdemo.pg-datastore` "uses" the interface package `linkdemo.policy.datastore` where the generic functions we need to implement methods for are defined.

The PostgreSQL backend will be implemented in a new file called `pg-datastore.lisp`, lets add it to `linkdemo.asd`:

```
(asdf:defsystem #:linkdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML :POSTMODERN :IRONCLAD :BABEL)
  :components ((:file "defmodule")
               (:file "pg-datastore")
               (:file "linkdemo")))
```

Next, we create the file `pg-datastore.lisp` in the project directory and add an `in-package` declaration:

```
;;;; pg-datastore.lisp

(in-package #:linkdemo.pg-datastore)
```

The schema

The app will be very simple, it will have users, who can post links, and vote on them. That makes three tables:

- A `users` table with an id, username and password fields.
- A `links` table with an id, url, title, and submitter fields, where the submitter will be a foreign key to the `users` table.
- The third table will be called `votes` and we will store all of the upvotes, it will have two fields, both foreign keys to the link and the user who upvoted it.

We could have stored the upvotes as an integer in the `links` table, but then users would be able to vote more than once per link, and we don't want that. What we need in this case is a many to many relation. If you are familiar with the basics of relational databases, this would be the most straightforward way to model our data.

Connecting

There are two ways to connect to a PostgreSQL database, using the macro `with-connection` whose body will be executed in the context of a connection. Or using `connect-toplevel` which will create a connection and setup the special variable `*database*` to the new connection. This variable is used to execute queries. `with-connection` automatically binds it in its body. I'll be using `with-connection` for our code, but `connect-toplevel` is useful for testing at the REPL so we don't have to wrap all of our queries in `with-connection`.

In order to use the macro, we'll need to have a variable with the connection spec, which has the following form: `(database user password host)`. The connection spec will be stored in a slot of our `pg-datastore` class(the one used for dispatch). Let's define this class in `pg-datastore.lisp`:

```
(defclass pg-datastore ()
  ((connection-spec :initarg :connection-spec
                    :accessor connection-spec)))
```

For testing purposes, I'll create an instance of this class and store it in a variable called `*db*`(in our real app, we'll use `*datastore*` in the internal package):

```
(defparameter *db*
  (make-instance 'pg-datastore
                 :connection-spec '("linkdemo" "linkdemouser" "mypass" "localhost")))
```

We can now do this:

```
(with-connection (connection-spec *db*)
  ;query goes here
  )
```

Defining the tables.

DAO classes

Postmodern isn't an ORM, so if you're used to them from places like Django or Rails, you're in luck, because in my very humble opinion, they suck. If you're using a RDBMS, learn the relational model already. Now that I got that rant out of the way, lets move on. Even though it isn't an ORM, postmodern does allow us to work with objects, but they are just simple DAOs(Database Access Objects). DAO objects are defined the same way ordinary lisp objects are, using `defclass`, but they have additional syntax for giving database types to our slots, and we need to add the `dao-class` [2] to the definition. Here is how the `users` table will be defined:

```
(defclass users ()
  ((id :col-type serial :reader user-id)
   (name :col-type string :reader user-name :initarg :name)
   (password :col-type string :reader user-password :initarg :password)
   (salt :col-type string :reader user-salt :initarg :salt))
  (:metaclass dao-class)
  (:keys id))
```

The difference between a standard class definition and a dao class is that we have a `:col-type` option to slot definitions that specify what database type we want to create. In our case, `id` will be a `serial` which is the PostgreSQL type for an integer that will auto-increment every time we add a record. The other two fields will be strings. In order to add the `:col-type` option to our slots, as well as other additions to our dao classes we must specify `dao-class` as a metaclass. Metaclasses are the standard Common Lisp mechanism for extending the object system. We also specify that we want `id` to be a primary key. The `password` and `salt` slots will contain the password hash and salt from encrypting the password of the user.

We can see what SQL code will be generated by this definition with `dao-table-definition`:

```
 * (dao-table-definition 'users)
```

It will give us the following output as a string(formatting by me):

```
CREATE TABLE users (
  id SERIAL NOT NULL,
  name TEXT NOT NULL,
  password TEXT NOT NULL,
  salt TEXT NOT NULL,
  PRIMARY KEY (id)
)
```

Lets implement the method used for initiating the datastore, creating the tables seems like a good thing to put in it. The generic function is named `datastore-init`, here it is:

[2] A metaobject protocol is a scary term which basically means that even classes are instances of a class. If you aren't familiar with object-oriented meta-programming, a metaclass controls the way other classes behave. In our case Postmodern provides us a metaclass that tells CLOS classes how to be saved into a database. [end of oversimplified footnote]

```
(defmethod datastore-init ((datastore 'pg-datastore))
  (with-connection (connection-spec datastore)
    (unless (table-exists-p 'users)
      (execute (dao-table-definition 'users)))))
```

First we connect to the database, then, using the `table-exists-p` predicate we check if the table is already defined. If it isn't, we use the `execute` function, which will execute an SQL expression, in our case, it will be the output of `dao-table-definition`. Later we'll augment this method with the definitions of the other tables.

We can call this method like this:

```
(datastore-init *db*)
```

After the table is defined, we can add users by instantiating objects of the `users` class, and inserting them into the db using `insert-dao`, here is an example:

```
(with-connection (connection-spec *db*)
  (insert-dao (make-instance 'users
                             :name "user"
                             :password "pass")))
```

Querying

Say we've added a bunch of users to the db, we can now query them in two ways, as DAOs, or as an ordinary table. The dao way is with `select-dao`, which returns a list of lisp objects:

```
 * (with-connection (connection-spec *db*)
     (select-dao 'users ))

 => (#<USERS {10089B4443}> #<USERS {10089B63F3}> #<USERS {10089B69F3}>
 #<USERS {10089B6FF3}>)
```

We can also use a normal query using `S-SQL`, a lispy syntax for SQL. Have a look at the example(the password and salt values are made up of course):

```
 * (with-connection (connection-spec *db*)
     (query (:select :* :from 'users)))

 => ((1 "user" "pass" "salt")
     (2 "user1" "pass1" "salt1")
     (3 "user2" "pass2" "salt2")
     (4 "user3" "pass3" "salt3"))
```

The `query` form takes an `S-SQL` expression. S-sql operators are keywords. Our query returns a list of lists, with the values in the table. We can get slightly more useful output with the `query` `args/format` optional parameter which specifies the format of the result. The most common values are `:plists` and `:alists`, returning the result in the format of a plist or alist, with the column names. Example:

```
 * (with-connection (connection-spec *db*)
     (query (:select :* :from 'users) :plists))

 => ((:ID 1 :NAME "user" :PASSWORD "pass" :SALT "salt")
     (:ID 2 :NAME "user1" :PASSWORD "pass1" :SALT "salt1")
     (:ID 3 :NAME "user2" :PASSWORD "pass2" :SALT "salt2")
     (:ID 4 :NAME "user3" :PASSWORD "pass3" :SALT "salt3"))

 * (with-connection (connection-spec *db*)
     (query (:select :* :from 'users) :alists))

 => (((:ID . 1) (:NAME . "user") (:PASSWORD . "pass") (:SALT . "salt"))
     ((:ID . 2) (:NAME . "user1") (:PASSWORD . "pass1") (:SALT . "salt1"))
     ((:ID . 3) (:NAME . "user2") (:PASSWORD . "pass2") (:SALT . "salt2"))
     ((:ID . 4) (:NAME . "user3") (:PASSWORD . "pass3") (:SALT . "salt3")))
```

We'll see other format examples later on.

Links and votes

Because `defclass` definitions of dao objects don't support adding foreign keys, well have to use a slightly different method of defining tables using the `deftable` macro. We start off by defining our DAO:

```
(defclass links ()
  ((id :col-type serial :reader link-id)
   (url :col-type string :reader link-url :initarg :url)
   (title :col-type string :reader link-title :initarg :title)
   (submitter-id :col-type integer :reader link-submitter-id :initarg :submitter-id))
  (:metaclass dao-class)
  (:keys id))
```

Next, because we need to add the foreign key constrain, we use the `deftable` macro to define a table. The table will inherit all of the fields of the dao class:

```
(deftable links
  (!dao-def)
  (!foreign 'users 'submitter-id 'id))
```

`!dao-def` tells `deftable` to inherit the field definitions from the dao class definition, and `!foreign` tells `deftable` to add a foreign key constrain to the table. `!foreign`s first parameter is the target table, the second is the field, and if the field has a different name in the definition of the target table, add it as a third parameter.

Lets do the same for `votes`:

```
(defclass votes ()
  ((link-id :col-type integer :reader vote-link-id :initarg :link-id)
   (submitter-id :col-type integer :reader vote-submitter-id :initarg :submitter-id))
  (:metaclass dao-class)
  (:keys link-id submitter-id))

(deftable votes
  (!dao-def)
  (!foreign 'links 'link-id 'id)
  (!foreign 'users 'submitter-id 'id))
```

Now, let's update the `datastore-init` method to create these tables as well. Note that unlike ordinary dao-defined tables, tables defined with `deftable` are created in the database using the function `create-table`:

```
(defmethod datastore-init ((datastore pg-datastore))
  (with-connection (connection-spec datastore)
    (unless (table-exists-p 'users)
      (execute (dao-table-definition 'users)))
    (unless (table-exists-p 'links)
      (create-table 'links))
    (unless (table-exists-p 'votes)
      (create-table 'votes))))
```

Defining our interface

What will the interface consist of? We'll need a way to register a user, and authenticate one at login. We'll also need to be able to post a link and upvote it. Also we'll need a way to get a list of all the links for the home page, and a way to get their score, since we'll want to sort by it. That's about it for a quick version 1.

Hashing passwords

The original version of this chapter stored passwords in plain text, I decided to actually try to be secure in this revision. For this purpose I'll use the ironclad cryptography library to hash passwords. We'll use the pbkdf2 algorithm to hash our passwords:

```
(defun hash-password (password)
  (multiple-value-bind (hash salt)
      (ironclad:pbkdf2-hash-password (babel:string-to-octets password))
    (list :password-hash (ironclad:byte-array-to-hex-string hash)
          :salt (ironclad:byte-array-to-hex-string salt))))
```

This code is kind of dense, all you need to know about it is that it returns a plist with a password hash and a salt, ready to be stored into a database.

Checking to see if a password matches involves taking said password, hash and salt, hashing the password using the salt, and comparing hashes:

```
(defun check-password (password password-hash salt)
  (let ((hash (ironclad:pbkdf2-hash-password
               (babel:string-to-octets password)
               :salt (ironclad:hex-string-to-byte-array salt))))
    (string= (ironclad:byte-array-to-hex-string hash)
             password-hash)))
```

With this out of the way, we can now go on and write the user handling logic.

Handling users

When we create and authenticate a user we'll need a way to find if a user already exists in the database, `datastore-find-user` does this and returns a plist with the users credentials, and `nil` if no such user exists:

```
(defmethod datastore-find-user ((datastore pg-datastore) username)
  (with-connection (connection-spec datastore)
    (query (:select :* :from 'users
                    :where (:= 'name username))
           :plist)))
```

Note that the argument to query is `:plist` and not the plural `:plists`. This tells postmodern to return just one result.

Next, when a user logs in, we simply find the user, and check if the password matches. If so, we return the username. If no such user exists or the passwords don't match, we return nil:

```
(defmethod datastore-auth-user ((datastore pg-datastore) username password)
  (let ((user (datastore-find-user datastore username)))
    (when (and user
               (check-password password (getf user :password)
                                        (getf user :salt)))
      username)))
```

And finally registering the user. We check if the user is registered, and if not, we create a record in the db:

```
(defmethod datastore-register-user ((datastore pg-datastore) username password)
  (with-connection (connection-spec datastore)
    (unless (datastore-find-user datastore username)
      (let ((password-salt (hash-password password)))
        (when
            (save-dao
             (make-instance 'users
                            :name username
                            :password (getf password-salt :password-hash)
                            :salt (getf password-salt :salt)))
          username)))))
```

We check to see if the user isn't registered if he isn't, we hash the password, make a DAO object with the username, hash and salt, and save it. The reason `save-dao` is wrapped in a ‘when‘ is to make sure the operation was successful, if so, we return the username.

Handling links

In order to handle links properly, let's write an `datastore-upvoted-p` predicate method:

```
(defmethod datastore-upvoted-p ((datastore pg-datastore) link-id user)
  (with-connection (connection-spec datastore)
    (query (:select :link-id :users.name :from 'votes 'users
                    :where (:and (:= 'users.id 'submitter-id)
                                 (:= 'users.name user)
                                 (:= 'link-id link-id)))
           :plist)))
```

This is a slightly more complicated query, it even has an implicit join. Essentially, we query the `votes` table for any rows with the link-id and submitter-id matching that of the username of the user, and we return the link-id and the username.

Upvoting a link involves finding the id of the user, checking if the user hasn't already upvoted that link, and then simply doing an insert:

```
(defmethod datastore-upvote ((datastore pg-datastore) link-id user)
  (with-connection (connection-spec datastore)
    (let ((submitter-id (getf (datastore-find-user datastore user) :id)))
      (when (and submitter-id
                 (not (datastore-upvoted-p datastore link-id user)))
        (when (save-dao (make-instance 'votes
                                       :link-id link-id
                                       :submitter-id submitter-id))
          link-id)))))
```

For posting a link, we want the user submitting it to also automatically upvote it. In order to do that though, we have to use a DAO, since `query` will not return us the inserted value, and we need the links id in order to upvote it. After we `save-dao` the DAO though, its `link-id` accessor function will return it for us and we can upvote it. This is how it looks like:

```
(defmethod datastore-post-link ((datastore pg-datastore) url title user)
  (with-connection (connection-spec datastore)
    (let* ((submitter-id (getf (datastore-find-user datastore user) :id))
           (link (make-instance 'links
                                :url url
                                :title title
                                :submitter-id submitter-id)))
      (save-dao link)
      (datastore-upvote datastore (link-id link) user))))
```

Getting all the links involved 3 steps, selecting them, checking their upvote count, and then sorting them. We'll need to write a bunch of functions to do so. First let's write a function that selects them all:

```
(defun get-all-links/internal ()
  (query (:select :* :from 'links) :plists))
```

This function doesn't have a `with-connection` in its body because it is internal and will only be used in a context that has a connection.

Now, given a `link-id`, getting the upvote count is as easy as using the sql `COUNT` function. We tell `query` to format the result with the `:single` keyword, which returns a single result, in our case an integer:

```
(defmethod datastore-upvote-count ((datastore pg-datastore) link-id)
  (with-connection (connection-spec datastore)
    (query (:select (:count link-id) :from 'votes
                    :where (:= link-id 'link-id))
           :single)))
```

We'll need to augment the link plist with two keys: `:votes` is the number of votes the link has, and `voted-p` is a boolean specifying whether or not the currently logged in user has upvoted it. We do this for every link returned by `get-all-links/internal`. Let's define a function to do that. We have to pass `datastore` to it because it will call `datastore-upvoted-p`:

```
(defun add-vote-count (datastore links username)
  (loop
     for link in links
     for id = (getf link :id)
     collect (list* :votes (datastore-upvote-count datastore id)
                    :voted-p (datastore-upvoted-p datastore id username)
                    link)))
```

The simplest way I found to get them sorted is using common lisps `sort` function:

```
(defun sort-links (links)
  (sort links #'>
        :key #'(lambda (link) (getf link :votes))))
```

And finally we define our method:

```
(defmethod datastore-get-all-links ((datastore pg-datastore)
                                    &optional username)
  (with-connection (connection-spec datastore)
    (sort-links
     (add-vote-count datastore
                     (get-all-links/internal)
                     (or username "")))))
```

Note that we use `or` to pass the optional value `username`, in case it is nil, we want to pass an empty string, since there might be a case where no user is logged in, and `upvoted-p` expects a string, and will choke on `nil`.

Exporting the interface

The only thing we'll need to export from this package is the `pg-datastore` classname. Lets do that in `defmodule.lisp`:

```
(defpackage #:linkdemo.pg-datastore
  (:use #:cl #:postmodern #:linkdemo.policy.datastore)
  (:export #:pg-datastore))
```

Conclusion

That's it for the DB layer for now, In the next chapter we'll start using it to finish our app, and then we'll augment it to use a different backend datastore.

Here are some links for the curious:

- PostgreSQL web site[3]
- Postmodern[4]
- Postmodern examples[5]
- Using policy based design in RESTAS(in Russian)[6]

[3]http://www.postgresql.org/

[4]http://marijnhaverbeke.nl/postmodern/

[5]https://sites.google.com/site/sabraonthehill/postmodern-examples

[6]http://archimag.lisper.ru/2012/12/21/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_policy-based_design_%D0%B2_RESTAS

6 Persistence part II: The UI

Introduction

In the last chapter we developed a simple database schema using PostgreSQL and wrote a layer to access it using postmodern and used restas policies to make it reusable with other datastores. In this chapter we'll develop a very simple UI just to show how to actually use this API we wrote so that in the next one we can write an implementation using other datastores and show how it all "just works". You won't see much new stuff here, We've done most of this already.

Set up

Lets set up sexml, like we did in the previous project. In the `defmodule.lisp` file, add the sexml initialization code:

```lisp
(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))
```

Next, we'll need to add two new files to our project, `util.lisp` will contain some useful functions used in the rest of the project, and `template.lisp` where we keep our sexml templates:

```lisp
(asdf:defsystem #:linkdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML :POSTMODERN :IRONCLAD :BABEL)
  :components ((:file "defmodule")
               (:file "pg-datastore")
               (:file "util")
               (:file "template")
               (:file "linkdemo")))
```

Create both of these files and put `(in-package #:linkdemo)` at their tops.

The templates

Of course the first thing to do in the `template.lisp` file is to add:

```
(<:augment-with-doctype "html" "" :auto-emit-p t)
```

In order to get a doctype declaration in our output.

The first template is `html-frame`. The function takes a plist called `context` which will contain the data to be displayed, and creates an html page with a title and a menu with links:

```
(defun html-frame (context)
  (<:html
   (<:head (<:title (getf context :title)))
   (<:body
    (<:div
     (<:h1 (getf context :title))
     (<:a :href (genurl 'home) "Home") " | "
     (if (logged-on-p)
         (list (<:a :href (genurl 'submit) "Submit a link")
               " | "
               (<:a :href (genurl 'logout)
                    (format nil "Logout ~A"
                            (logged-on-p))))
         (list (<:a :href (genurl 'login) "Log in")
               " or "
               (<:a :href (genurl 'register) "Register")))
     (<:hr))
    (getf context :body))))
```

After the `h1` title we have a menu composed of links. The first is a link to the home page, followed by a separator: `" | "`. Next, we check if the user is logged in, using the function `logged-on-p` which we will define in `util.lisp` in a minute. We display the rest of the menu depending on this predicate. If the user is logged in, we display a link to a submit page and a link to log out. If the user isn't logged in, we prompt to either log in, or register. Simple enough I hope.

The `logged-on-p` function is defined in `util.lisp`:

```
(defun logged-on-p ()
  (hunchentoot:session-value :username))
```

Notice that it always returns the name of the user, if one is logged in, so we can use it in displaying the username in `html-frame`.

The next template is the actual home page, it takes a list of links, which are plists. It then displays all the links on separate lines. On each line, is either an upvote link, or a * if the user isn't logged in, followed by a vote count, and the actual link. If the user has already upvoted a link, instead of an upvote link, a * is displayed:

```
(defun home-page (links)
  (loop
     for link in links
     collect
       (<:div (if (logged-on-p)
                  (if (getf link :voted-p)
                      "*"
                      (<:a :href (genurl 'upvote-link :id (getf link :id))
                           "upvote"))
                  "*")
              " "
              (getf link :votes)
              " "
              (<:a :href (getf link :url) (getf link :title)))))
```

The last three templates are just the login and registration forms, and the form to submit a link:

```
(defun login-form ()
  (<:form :action (genurl 'login/post) :method "post"
          "User name:" (<:br)
          (<:input :type "text" :name "username")(<:br)
          "Password:" (<:br)
          (<:input :type "password" :name "password") (<:br)
          (<:input :type "submit" :value "Log in")))

(defun register-form ()
  (<:form :action (genurl 'register/post) :method "post"
          "User name:" (<:br)
          (<:input :type "text" :name "username")(<:br)
          "Password:" (<:br)
          (<:input :type "password" :name "password") (<:br)
          (<:input :type "submit" :value "Register")))

(defun submit-form ()
  (<:form :action (genurl 'submit/post) :method "post"
          "Title:" (<:br)
          (<:input :type "text" :name "title") (<:br)
          "URL:" (<:br)
          (<:input :type "text" :name "url") (<:br)
          (<:input :type "submit" :value "Submit")))
```

I hope there isn't anything I need to explain here anymore.

The routes

Lets start with the home page:

```
(define-route home ("")
  (list :title "Linkdemo"
        :body (home-page (get-all-links (logged-on-p)))))
```

This is the first time we actually used the front end of our database layer. Restas has generated a convenient function called `get-all-links` that might look something like:

```
(defun get-all-links (&optional username)
  (datastore-get-all-links *datastore* username))
```

Remember that this method returned all the links, in a form specific to the logged in user, so that is why we pass in the username too. If there is no user, `datastore-get-all-links` handles that for us nicely. This all makes the code of the route very pretty.

Handling users

The next five routes handle user registration, logging in, and logging out. First, logging in:

```
(define-route login ("login")
  (list :title "Log in"
        :body (login-form)))
```

Now, lets handle the login form:

```
(define-route login/post ("login" :method :post)
  (let ((user (auth-user (hunchentoot:post-parameter "username")
                         (hunchentoot:post-parameter "password"))))
    (if user
        (log-in user)
        (redirect 'login))))
```

This route is fairly simple, we pass the post data to `auth-user`, which as we already said, will call the method `datastore-auth-user` and return a username if it is successful. We check if it is, and if so, we log the user in using the function `log-in`, otherwise we redirect the user to try again. Here is `log-in`s definition is util.lisp:

```
(defun log-in (username &optional (redirect-route 'home))
  (hunchentoot:start-session)
  (setf (hunchentoot:session-value :username) username)
  (redirect redirect-route))
```

It will start a session, set it to the username and redirect the user to a supplied route, `home` by default.

The code to register a user is almost exactly the same:

```
(define-route register ("register")
  (list :title "register"
        :body (register-form)))

(define-route register/post ("register" :method :post)
  (let ((user (register-user (hunchentoot:post-parameter "username")
                             (hunchentoot:post-parameter "password"))))
    (if user
        (log-in user)
        (redirect 'register))))
```

Finally, we have a route to log out the user:

```
(define-route logout ("logout")
  (log-out))
```

it just calls the `log-out` function, again, defined in `util.lisp`:

```
(defun log-out (&optional (redirect-route 'home))
  (setf (hunchentoot:session-value :username) nil)
  (redirect redirect-route))
```

Handling links

Submitting a link is just as easy as logging in, or registering:

```
(define-route submit ("submit")
  (list :title "Submit a link"
        :body (submit-form)))

(define-route submit/post ("submit" :method :post)
  (let ((link (post-link (hunchentoot:post-parameter "url")
                         (hunchentoot:post-parameter "title")
                         (logged-on-p))))
    (if link
        (redirect 'home)
        (redirect 'submit))))
```

A bit more interesting is the `upvote-link` route:

```
(define-route upvote-link ("upvote/:id")
  (:sift-variables (id #'parse-integer))
  (when (logged-on-p)
    (upvote id (logged-on-p)))
  (redirect 'home))
```

Getting it to run

In `util.lisp`, lets define a `start-linkdemo` function that will set up our application:

```
(defun start-linkdemo (&key
                        (port 8080)
                        (datastore 'linkdemo.pg-datastore:pg-datastore)
                        (datastore-init nil))
  (setf *datastore* (apply #'make-instance datastore datastore-init))
  (init)
  (start '#:linkdemo :port port :render-method 'html-frame))
```

Essentially we have 3 keyword parameters, a port, a datastore, by default it will be our PostgreSQL class, and `:datastore-init` which is the arguments to supply to `make-instance` when we instantiate the datastore class. In our case, that will be the connection spec. Let's export it in `defmodule.lisp`:

```
(restas:define-module #:linkdemo
  (:use #:cl #:restas #:linkdemo.datastore)
  (:export #:start-linkdemo))
```

Now our app is complete. Load it, and try it out at the repl:

```
 * (linkdemo:start-linkdemo
     :datastore-init '(:connection-spec ("linkdemo" "linkdemouser" "mypass" "localhost")))
```

Try it out, play with it. We'll continue in the next chapter.

7 Persistence part III: Redis

Introduction

We have already implemented a mostly complete application using PostgreSQL, now we'll implement the datastore backend using redis. Because of the separation between interface and implementation that CLOS and restas policies provide us we won't have to touch any of the application code outside of adding redis as a dependency and actually implementing the policy to use it.

Redis is a very simple key-value store, it has support for strings, sets, lists and hash tables, and uses a very simple protocol and api. It has no query language, and no way to define a schema, which both simplifies and complicates things.

I assume redis is installed and is running on the local machine. On debian-like systems, this is just an `apt-get install redis-server` away. You can see if redis is running by starting the console client `redis-cli` and typeing `ping` in the prompt, it should respond with "PONG".

Next, let's add `cl-redis`, the lisp client library for redis to our list of dependencies in `linkdemo.asd`. Also, lets add a file named `redis-datastore.lisp` to the project. This is how our system definition looks like:

```
(asdf:defsystem #:linkdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML :POSTMODERN :ironclad :babel :cl-redis)
  :components ((:file "defmodule")
               (:file "pg-datastore")
               (:file "redis-datastore")
               (:file "util")
               (:file "template")
               (:file "linkdemo")))
```

In `defmodule.lisp` we must add a package definition for our implementation of the redis backend:

```
(defpackage #:linkdemo.redis-datastore
  (:use #:cl #:redis #:linkdemo.policy.datastore)
  (:export #:redis-datastore))
```

Like `linkdemo.pg-datastore` this package uses the `cl` package, the `redis` package which contains the redis api and the policy package, where the names of the methods we must implement reside. We export the symbol `redis-datastore`, which names the class we use to access the datastore.

A note on redis

Redis commands are fairly simple, but some of them conflict with Common Lisp names, so they are all prefixed with `red-`. For instance getting the value of the key “foo” is done with `(red-get "foo")`. Connecting to the datastore is done with the `connect` function which takes `:host` and `:port` parameters or with the `with-connection` macro which takes the same parameters. Like with postmodern, we’ll be using `with-connection`.

The “schema” or lack there of

Because redis is schemaless, we must decide on a way to structure our data in redis. After some research and experimentation this is what I came up with:

A user record will be stored as a string, containing the printed representation of a plist. We’ll use the common lisp read/print consistency feature to ensure that what we print into that string will be read back consistently as a lisp list. Essentially, we’re serializing data into strings. Each such record will be kept under a key named `USER:{id}`, where id is an integer denoting the id of the record. Since keys are strings, we’ll have to generate them with a function using string concatenation. We’ll also need another record called `USER-IDS` which is an integer we’ll increment every time we add a user, and use the value as a new id.

Because we’ll also need to lookup users by their usernames, we’ll add another key to the datastore, called `USERNAME:{username}`, where `username` is the username string. This key will hold the id we’ll use to lookup the whole record for that user. This is a sort of reverse lookup if you will.

Posts we’ll store in a similar way, we’ll have a `POSTS-IDS` key with the id count, and we’ll have a `POST:{id}` key holding the serialized plist record for that post.

Upvotes are interesting because we can just use sets to store the set of user names that have upvoted a given post. We’ll store these sets in keys named `UPVOTE:{post-id}`.

So for example if we have one user, named `user`, the datastore will contain this information:

```
"USER-IDS" : 1
"USER:1" : "(:id 1 :username \"user\" :password \"...\" :salt \"...\")"
"USERNAME:user" : 1:
```

If he posts a link, it would look like this:

```
"POST-IDS" : 1
"POST:1" : "(:id 1 :url \"http://google.com\" :title \"google\" :submitter-id 1)"
```

And the corresponding upvote:

```
"upvote:1" : {"user", }
```

The implementation

In `redis-datastore.lisp` first lets define our datastore access class. Redis connections take two arguments, a host and a port, by default they are the local host, and 6379, Here is the class:

```lisp
(in-package #:linkdemo.redis-datastore)

(defclass redis-datastore ()
  ((host :initarg :host :initform #(127 0 0 1) :accessor host)
   (port :initarg :port :initform 6379 :accessor port)))
```

Note that the host is given as a vector denoting the ip address `127.0.0.1`. Since there is nothing to initialize, the `datastore-init` method is empty:

```lisp
(defmethod datastore-init ((datastore redis-datastore)))
```

Convenience functions

Next are a couple of convenience functions we'll need, first are the familiar hash-password and check-password from the pg-datastore.lisp file. Take it as an exercises to move these functions to a separate package in a separate file and eliminate the duplication, if it bugs you:

```lisp
(defun hash-password (password)
  (multiple-value-bind (hash salt)
      (ironclad:pbkdf2-hash-password (babel:string-to-octets password))
    (list :password-hash (ironclad:byte-array-to-hex-string hash)
          :salt (ironclad:byte-array-to-hex-string salt))))

(defun check-password (password password-hash salt)
  (let ((hash (ironclad:pbkdf2-hash-password
               (babel:string-to-octets password)
               :salt (ironclad:hex-string-to-byte-array salt))))
    (string= (ironclad:byte-array-to-hex-string hash)
             password-hash)))
```

In order to store lisp lists in redis, we need to print and read them consistently. Fortunately for us, lisp is itself a kind of serialization format. Lisp objects like symbols, keywords, strings, lists and numbers can be printed to a string, and then read back as lisp objects. Here are two functions that will do that for us:

```lisp
(defun serialize-list (list)
  (with-output-to-string (out)
    (print list out)))

(defun deserialize-list (string)
  (let ((read-eval nil))
    (read-from-string string)))
```

And finally, we need a way to generate keys like "USER:1" and "USER:2" and so on, the function `make-key` takes a keyword and a string or number and generates a key for us:

```
(defun make-key (prefix suffix)
  (format nil "~a:~a" (symbol-name prefix) suffix))
```

Handling users

In order to get a user record by username, first we lookup the user id, We do this simply with a `red-get` command and a key in the form of `"USERNAME:{username}"`, generated with `make-key`. Next we use the id to retrieve the user record and convert it to a list with `deserialize-list`:

```
(defmethod datastore-find-user ((datastore redis-datastore) username)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (let ((user-id (red-get (make-key :username username))))
      (when user-id
        (deserialize-list (red-get (make-key :user user-id)))))))
```

Authenticating the user is done with almost identical code to the postmodern example:

```
(defmethod datastore-auth-user ((datastore redis-datastore) username password)
  (let ((user (datastore-find-user datastore username)))
    (when (and user
               (check-password password
                               (getf user :password)
                               (getf user :salt)))
      username)))
```

Registering the user on the other hand is a bit more involved. First we must create a new id by using the `red-incr` command, which increments the value of the `USER-IDS` key. Then, we must use this id to generate a new “USERS:{id}” key, and set it to the serialized plist containing the user information. We must then add the id as a value to the “USERNAME:{username}” key. And finally, we return the username. Not to forget also hashing the password, and checking if such a user already exists:

```
(defmethod datastore-register-user ((datastore redis-datastore) username password)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (unless (datastore-find-user datastore username)
      (let* ((password-salt (hash-password password))
             (id (red-incr :user-ids))
             (record (list :id id
                           :username username
                           :password (getf password-salt :password-hash)
                           :salt (getf password-salt :salt))))
        (red-set (make-key :user id) (serialize-list record))
        (red-set (make-key :username username) id)
        username))))
```

Handling upvotes

Checking if a user has upvoted a link is as easy as checking to see if that user is in the set of upvoters for that link. Sets in redis are accessed with the `red-sismember` command(which I assume stands for “set is member”). It simply takes a key and a value and checks to see if that value is in the set denoted by the key:

```
(defmethod datastore-upvoted-p ((datastore redis-datastore) link-id user)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (red-sismember (make-key :upvote link-id) user)))
```

Upvoting a post is also a fairly simple task. First we must check if the user exists, and that the link isn’t upvoted, and then we simply add the username to the set of users who have upvoted this link. Adding an element to a set is done with the `red-sadd` command:

```
(defmethod datastore-upvote ((datastore redis-datastore) link-id user)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (when (and (datastore-find-user datastore user)
               (not (datastore-upvoted-p datastore link-id user)))
      (when (red-sadd (make-key :upvote link-id) user)
        link-id))))
```

Handling links

Posting a link involves first getting the user id of the submitter, generating a new id for the link, and then setting the key “POST:{id}” to the serialized plist of the record. After that we upvote the link:

```
(defmethod datastore-post-link ((datastore redis-datastore) url title user)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (let* ((submitter-id (getf (datastore-find-user datastore user) :id))
           (id (red-incr :posts-ids))
           (link (list :id id
                       :url url
                       :title title
                       :submitter-id submitter-id)))
      (red-set (make-key :post id) (serialize-list link))
      (datastore-upvote datastore (getf link :id) user))))
```

Extracting all the links is a bit interesting. Somehow we bust get a list of all keys that start with “POST:” and then extract them all. We’re in luck, since redis has a command `red-keys` that returns a list of keys matching a pattern, we simply pass it “POST:*” and we’ll get them all. Then we “get” the keys and deserialize their values:

```
(defun get-all-links/internal ()
  (let ((keys (red-keys (make-key :post "*"))))
    (loop for key in keys
          collect (deserialize-list (red-get key)))))
```

Getting the upvote count is as easy as counting the elements of a set, and fortunate for us, redis has such a command, `red-scard`, which I can never remember, and always have to lookup :)

```
(defmethod datastore-upvote-count ((datastore redis-datastore) link-id)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (red-scard (make-key :upvote link-id))))
```

The functions `add-vote-count`, `sort-links` and `datastore-get-all-links` are almost the same:

```
(defun add-vote-count (datastore links username)
  (loop
     for link in links
     for id = (getf link :id)
     collect (list* :votes (datastore-upvote-count datastore id)
                    :voted-p (datastore-upvoted-p datastore id username)
                    link)))

(defun sort-links (links)
  (sort links #'>
        :key #'(lambda (link) (getf link :votes))))

(defmethod datastore-get-all-links ((datastore redis-datastore) &optional username)
  (with-connection (:host (host datastore)
                    :port (port datastore))
    (sort-links
     (add-vote-count datastore
                     (get-all-links/internal)
                     (or username "")))))
```

Conclusion

And we're done. Save the file, reload the code and now we can start our redis backed app with:

```
* (linkdemo:start-linkdemo
    :datastore 'linkdemo.redis-datastore:redis-datastore)
```

And in fact, if we swap out the value of `*datastore*` with an instance of `redis-datastore` or `pg-datastore` we can switch the backend database while the app is running. This is pretty cool IMHO.

Further reading:

- Redis home page[1]
- List of Redis commands[2]
- The Beauty of Simplicity: Mastering Database Design Using Redis by Ryan Briones(video)[3]
- The Beauty of Simplicity: Mastering Database Design Using Redis by Ryan Briones(slides)[4]

[1]http://www.redis.io/
[2]http://www.redis.io/commands
[3]http://vimeo.com/26385026
[4]http://www.slideshare.net/ryanbriones/the-beauty-of-simplicity-mastering-database-design-with-redis

8 Modules as reusable components part I: restas-directory-publisher

Introduction

With the way we've been using Restas so far, you probably think of modules defined with `restas:define-module` as simply a way to package routes together, but they are much more than that. A module is a tool for building reusable and composable components of a web app. A module can act both as a stand alone web app, or a component of a larger site. A module can even be used more than once in the same app, for different, but similar purposes. In the following chapters, we'll see what modules have to offer, and how to use them. I'll start by showing you how to use one that already exists in your web app.

`restas-directory-publisher` is a module designed to handle serving of static files and directories. It can be used to serve a static website, or just handle your "static/" directory containing css and javascript files. Although static files should usually be served by your frontend server like nginx, for testing during development, and for low-traffic sites `restas-directory-publisher` gets the job done.

Handling static files in restas

In general, if you want to serve a static file in restas, all you have to do is have your route return a pathname object, for instance, serving a css file for your app could be done in such a way:

```
(define-route css ("static/style.css" :content-type "text/css")
  #P"/path/to/style.css")
```

This works, but if we have many static files, we have to define routes for every one. And if we add or remove some of them on a regular basis, it gets even more tedious to do so. This is the problem `restas-directory-publisher` solves.

Mounting modules

In order to explain what it means for a module to be mounted, let me make a loose analogy with object oriented programming. In classic OOP you have a class that defines the properties and methods of a class. It serves as a template for the actual object. In order to get one such object, you must "instantiate" your class. Modules work in a similar way, before you mount it, a module is just a regular common lisp package. After you mount it, they become "alive" so to speak, as a running app. The function `restas:start` takes a module name, and mounts it at the top level, but a module can also have submodules mounted in it, forming a sort of hierarchy.

For example you can have a main module, and this module has several submodules, for instance a blog module, a login module to handle user authentication, and an admin module for updating the blog. Each of these modules might have submodules of their own. The main module will be mounted using `restas:start`, as we've seen already, but the submodules will be mounted with `restas:mount-module`.

`restas:mount-module` has a simple syntax, it takes a symbol to name the mount, and the name of a module in parentheses. Additionally, it takes more options which we will take a look at later. Here is how our blog example structure will be implemented:

```
(in-package #:blogapp.main)

(mount-module -login- (#:blogapp.login)
  (:url "login"))

(mount-module -blog- (#:blogapp.blog)
  (:url "blog"))

(mount-module -admin- (#:blogapp.admin)
  (:url "admin"))

(start '#:blogapp.main :port 8080)
```

Pretty much the only thing I need to explain here is the `:url` option. It specifies the url where the module must be mounted. Take the `#:blogapp.blog` module as an example, lets say it has a route with the following template `"post/:id"`, if we mount it under the `"blog"` url, as we've done, the template will now be transformed to `"blog/post/:id"`, and the route will match that url. If we leave out the `:url` option, the routes in the submodule will be treated as if they were top level.

Adding a stylesheet to linkdemo

Because `linkdemo` is a project defined with `restas-project`, it has a directory called `static/`, we'll define a file called `static/css/style.css` and use `restas-directory-publisher` in order to serve it.

First, we have to add `restas-directory-publisher` as a dependency to the `linkdemo.asd` file:

```
(asdf:defsystem #:linkdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML :POSTMODERN :ironclad :babel :cl-redis :restas-directory-pub\
lisher)
  :components ((:file "defmodule")
               (:file "pg-datastore")
               (:file "redis-datastore")
               (:file "util")
               (:file "template")
               (:file "linkdemo")))
```

Next, lets add the file `static/css/style.css` to our templates. In `template.lisp`, edit `html-frame`:

```
(defun html-frame (context)
  (<:html
   (<:head (<:title (getf context :title))
           ;; Stylesheet
           (<:link :rel "stylesheet" :type "text/css" :href "/static/css/style.css"))
   (<:body
    (<:div
     (<:h1 (getf context :title))
     (<:a :href (genurl 'home) "Home") " | "
     (if (logged-on-p)
         (list (<:a :href (genurl 'submit) "Submit a link")
               " | "
               (<:a :href (genurl 'logout)
                    (format nil "Logout ~A"
                            (logged-on-p))))
         (list (<:a :href (genurl 'login) "Log in")
               " or "
               (<:a :href (genurl 'register) "Register")))
     (<:hr))
    (getf context :body))))
```

Now, create the file `static/css/style.css`:

```
h1 {
  font-family: "Helvetica";
  color: #c44cc4;
}
```

And finally, add the following code to the bottom of `defmodule.lisp` to add the `mount-module` declaration:

```
(mount-module -static- (#:restas.directory-publisher)
  (:url "static")
  (restas.directory-publisher:*directory* *static-directory*))
```

Other than the `:url` declaration, we have a variable binding. The variable `restas.directory-publisher:*directory*` signifies where `restas-directory-publisher` should look for static files to serve, we give it the value of `linkdemo::*static-directory*` which `restas-project` created for us in `defmodule.lisp`.

Contexts

A set of such variables, like `restas.directory-publisher:*directory*` exported from a module, and later bound when mounting a module are called a context. One way to explain what a module context is, is to expand on the OO metaphor. When me "instantiate" a module with `mount-module`, we set the current "instance" variables. If we were to mount the same module more than once(let's say we have more than one static directory we want to serve) we would give the two mounts different values. Although there is just one variable(in the lisp sense) called `restas.directory-publisher:*directory*`, at request time Restas will bind it to the right context value depending on which mounted module is handling the request.

In other words, a context is just a set of `(defvar *foo* ...)` inside a module, that are being rebound depending on which mounted module the route that handles the request belongs to. I hope I haven't confused you. This mechanism gives us a lot of power to reuse and configure modules.

Conclusion

The Restas module mechanism is very powerful. Without them, restas can be considered only slightly more powerful than any other micro framework you might have seen. This makes it a powerful and useful tool for web development. If I am permitted a very short rant, this as well as the policy mechanism are both examples of the power of dynamic variables in Lisp. Even though an FP proponent will denounce them because they break referential transparency, they are a really powerful tool, and I miss them every time I have to use a language that doesn't have them. Even something as simple as binding a `*request*` variable to the current request object in hunchentoot is a great plus.

In the next chapters we'll examine modules in a bit more depth with a few examples. In the mean time, happy hacking!

Some links:

- OUTDATED(but might give you a sense of the general idea): Restas docs on modules[1]
- Blog post explaining the new Restas module system(IN RUSSIAN): Изменение системы модулей в RESTAS[2]

[1]http://restas.lisper.ru/en/manual/modules.html

[2]http://archimag.lisper.ru/2013/01/02/%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D0%B5%D0%B9_%D0%B2_RESTAS

9 Modules as reusable components part II: Implementing a reusable module

Introduction

In this chapter we'll refactor the user handling code from `linkdemo` out into it's own project, which we can then reuse in other apps as well. First, let's state explicitly how will the authentication module interact with the rest of the system:

- I want the module to know as little as possible about how users are actually represented. What this means is that the module will be passed references to functions that will handle user creation and authentication. Essentially these will be context variables set on module initialization. What we get from this is that we won't have to worry about implementing a policy or even knowing that one exists. We just call functions that the user of the module gave us.
- The linkdemo application will need a way to ask the auth module if a user is logged in.
- The linkdemo application will also need a way to generate links to the login, logout and registration routes. I'll explain later how Restas solves this problem for us.
- The auth module will also need to know where to redirect the user once a successful operation has occurred. For example the user gets redirected to the home page after a successful login.

Lets start by creating a project:

```
* (ql:quickload "restas-project")
* (restas-project:start-restas-project "authdemo" :depends-on '(:sexml))
```

The interface

Now, let's define the endpoints where the module will be configured. This basically consists of a couple of `defvars` at the top of `authdemo.lisp`:

```
;;;; authdemo.lisp

(in-package #:authdemo)

;;; "authdemo" goes here. Hacks and glory await!

(defvar *authenticate-user-function* nil)
(defvar *register-user-function* nil)
(defvar *redirect-route* nil)
```

In order to keep things simple, when a user logs in, out or registers, we'll redirect to the same place, specified by `*redirect-route*`. We could have had separate variables for all 3 operations if we wanted.

The templates

Now let's port the templates from linkdemo. First, let's add the `template.lisp` file to the `authdemo.asd` file:

```
(asdf:defsystem #:authdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML)
  :components ((:file "defmodule")
               (:file "template")
               (:file "authdemo")))
```

Next, we add the `sexml` initialization code to`defmodule.asd` as well as adding `#:restas` to the `:use` list of the `#:authdemo` package:

```
;;;; defmodule.lisp

(restas:define-module #:authdemo
  (:use #:cl #:restas))

(in-package #:authdemo)

(defparameter *template-directory*
  (merge-pathnames #P"templates/" authdemo-config:*base-directory*))

(defparameter *static-directory*
  (merge-pathnames #P"static/" authdemo-config:*base-directory*))

(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))
```

Now we need to actually port the templates to `template.lisp`. The only functions we need are `login-form` and `register-form`, the code is exactly the same:

```
;;;; template.lisp

(in-package #:authdemo)

(defun login-form ()
  (<:form :action (genurl 'login/post) :method "post"
          "User name:" (<:br)
          (<:input :type "text" :name "username")(<:br)
          "Password:" (<:br)
          (<:input :type "password" :name "password") (<:br)
          (<:input :type "submit" :value "Log in")))

(defun register-form ()
  (<:form :action (genurl 'register/post) :method "post"
          "User name:" (<:br)
          (<:input :type "text" :name "username")(<:br)
          "Password:" (<:br)
          (<:input :type "password" :name "password") (<:br)
          (<:input :type "submit" :value "Register")))
```

The routes

In order to implement the routes, first we need to port the util functions from `linkdemos` `util.lisp` file. I'll keep them in `authdemo.lisp` for simplicity:

```
(defun logged-on-p ()
  (hunchentoot:session-value :username))

(defun log-in (username &optional (redirect-route *redirect-route*))
  (hunchentoot:start-session)
  (setf (hunchentoot:session-value :username) username)
  (redirect redirect-route))

(defun log-out (&optional (redirect-route *redirect-route*))
  (setf (hunchentoot:session-value :username) nil)
  (redirect redirect-route))
```

The only thing changed here is the default value of the `redirect-route` parameter of `log-in` and `log-out` is the `*redirect-route*` variable.

And finally, let's port the routes to `authdemo.lisp` routes:

```lisp
(define-route login ("login")
  (list :title "Log in"
        :body (login-form)))

(define-route login/post ("login" :method :post)
  (let ((user (funcall *authenticate-user-function*
                       (hunchentoot:post-parameter "username")
                       (hunchentoot:post-parameter "password"))))
    (if user
        (log-in user)
        (redirect 'login))))

(define-route register ("register")
  (list :title "register"
        :body (register-form)))

(define-route register/post ("register" :method :post)
  (let ((user (funcall *register-user-function*
                       (hunchentoot:post-parameter "username")
                       (hunchentoot:post-parameter "password"))))
    (if user
        (log-in user)
        (redirect 'register))))

(define-route logout ("logout")
  (log-out))
```

The difference here is that in `login/post` and `register/post`, instead of calling `auth-user` and `register-user` directly, we use `funcall` to call the functions we've configured the module to use with the variables `*authenticate-user-function*` and `*register-user-function*`.

And finally, we must export `logged-on-p` since the applications using our module will need it:

```lisp
;;;; defmodule.lisp

(restas:define-module #:authdemo
  (:use #:cl #:restas)
  (:export #:logged-on-p))
```

And that's it! The module is now a complete reusable component! In order to use it though, we'll need to clear all of the things we ported to `authdemo` from `linkdemo`. Do that as an exercise and we'll move on to integrating them together.

Using authdemo in an application

getting linkdemo ready

Now we're ready to actually add `authdemo` to our link sharing application. First we need to add it to the dependencies in `linkdemo.asd`:

```
(asdf:defsystem #:linkdemo
  :serial t
  :description "Your description here"
  :author "Your name here"
  :license "Your license here"
  :depends-on (:RESTAS :SEXML :POSTMODERN :IRONCLAD
               :BABEL :cl-redis :restas-directory-publisher :authdemo)
  :components ((:file "defmodule")
               (:file "pg-datastore")
               (:file "redis-datastore")
               (:file "util")
               (:file "template")
               (:file "linkdemo")))
```

Next, we need to add `#:authdemo` to the `:use` list in the `linkdemo` package declaration so we can have access to the `logged-on-p` function:

```
(restas:define-module #:linkdemo
  (:use #:cl #:restas #:linkdemo.datastore #:authdemo)
  (:export #:start-linkdemo))
```

Mounting everything

Next, let's mount `authdemo` in our application. At the bottom of `defmodule.lisp` add the following code:

```
(mount-module -authdemo- (#:authdemo)
  (:render-method 'html-frame)
  (authdemo:*authenticate-user-function* #'auth-user)
  (authdemo:*register-user-function* #'register-user)
  (authdemo:*redirect-route* 'home))
```

We give our mount the name `-authdemo-`, and we specify a render method, in our case `html-frame`. Next, we bind the context variables in `authdemo`. We specify that the `home` route is to be used as the redirect route.

On routes in mounted modules

Before we continue I need to expand a bit on what happens when a module gets mounted. Consider the following example, we have a module named `foobar` with two routes in it, `foo` and `bar`. If we mount this module like so:

```
(mount-module -foobar- (#:foobar)
  ...)
```

Restas will automatically generate symbols for every route in `foobar` in the current package. In our case these symbols would be `-foobar-.foo` and `-foobar-.bar`. We can now use these symbols with `genurl` or `redirect` from a different module. Now modules can interact without knowing much about each other. We'll use this to "link" `linkdemo` and `authdemo`. Note that sub-modules can still use their parents routes, which is why we can pass just `home` as a redirect route to `authdemo`. But we'll need these auto-generated route symbols in the next section.

Fixing the templates

Because the `login`, `register` and `logout` routes are no longer available in the `linkdemo` application, we need to fix the template code that use them to generate links. Fortunately, only `html-frame` uses them, so fixing the template is easy, here is the fixed code:

```
(defun html-frame (context)
  (<:html
   (<:head (<:title (getf context :title))
           ;; Stylesheet
           (<:link :rel "stylesheet" :type "text/css" :href "/static/css/style.css"))
   (<:body
    (<:div
     (<:h1 (getf context :title))
     (<:a :href (genurl 'home) "Home") " | "
     (if (logged-on-p)
         (list (<:a :href (genurl 'submit) "Submit a link")
               " | "
               (<:a :href (genurl '-authdemo-.logout)
                    (format nil "Logout ~A"
                            (logged-on-p))))
         (list (<:a :href (genurl '-authdemo-.login) "Log in")
               " or "
               (<:a :href (genurl '-authdemo-.register) "Register")))
     (<:hr))
    (getf context :body))))
```

Notice that instead of `login`, we use `-authdemo-.login` as the route name, same for `register` and `logout`.

Running it

We can now run `linkdemo` like we did before:

```
 * (linkdemo:start-linkdemo
    :datastore 'linkdemo.redis-datastore:redis-datastore)
```

or if you prefer the PostgreSQL version:

```
(linkdemo:start-linkdemo
     :datastore-init '(:connection-spec ("linkdemo" "linkdemouser" "mypass" "localhost")))
```

Conclusion

And that's it! I've covered most of Restas by this point, as well as a bunch of other stuff like Postmodern and how to use Redis with lisp, html templating, etc. Now you know enough to write your own applications. Good luck, and have fun!

Appendix A: Getting started

Linux

Getting a Lisp implementation

The two implementations I recommend for use in this book are SBCL and CCL, both are very good, open source and generate fast code. If you are on windows or OS X, I recommend CCL, SBCL on Linux. I've had at least two people report to me problems with sbcl on OS X and and my tutorials, which is probably because of improperly built binaries, rather than an actual problem, but if you don't feel like compiling your SBCL from source(which I recommend), stick with CCL on those platforms for now.

Most Linux distributions have both CCL and SBCL in their package repositories, for example on Debian derived systems such as Ubuntu you can install sbcl with apt-get:

```
$ sudo apt-get install sbcl
```

But I recommend you download and install binaries manually, distributions sometimes patch CL implementations in order to "fix" something. Also who knows how ancient the version in the package manager is. It is usually recommended to work with the latest releases of CL implementations.

SBCL

You can download SBCL at http://www.sbcl.org/platform-table.html

Once you've done so, uncompress the archive. The example is shown for x86-64 on Linux:

```
$ tar -jxvf sbcl-1.1.5-x86-64-linux-binary.tar.bz2
```

Go to the directory:

```
$ cd sbcl-1.1.5-x86-64-linux/
```

The file `INSTALL` has information about how to configure the installation, but the default should suit your needs just fine, type:

```
$ sh install.sh
```

type `sbcl` into the command line to see if it works OK. you should get a prompt starting with an `*`. I have the habit of typeing `(+ 1 2)` in order to see if it really works, and I have never gotten an answer different than `3` so far, that's reliable software :)

CCL

You can get CCL from http://ccl.clozure.com/download.html The distribution contains both the 64 and 32 bit binaries. Chapter 2[1] of the CCL manual contains information on how to obtain and install CCL if you need it.

After you download CCL, uncompressed the archive with the following command:

```
$ tar -xzvf ccl-1.8-linuxx86.tar.gz
```

CCL is started by a shell script in the `ccl/scripts` directory, named `ccl` or `ccl64` for the 32 and 64 bit versions respectively. The way you install CCL is by copying one(or both) of these scripts to a directory on your path, and editing them to point to the CCL directory you just uncompressed. so for example if my `ccl` directory is in my home directory, named `/home/pav` on Linux:

```
$ sudo cp /home/pav/ccl/scripts/ccl64 /usr/local/bin
```

I then edit it to point to the `ccl` directory by setting the value of the variable `CCL_DEFAULT_DIRECTORY` at the top of the file to the `/home/pav/ccl/`.

Since I don't use the 32 bit version, I rename the file to simply `ccl`

```
$ sudo mv /usr/local/bin/ccl64 /usr/local/bin/ccl
```

I then ensure the file is executable:

```
$ sudo chmod +x /usr/local/bin/ccl
```

type `ccl` at the command line. The prompt should be `?`. Type some expression like `(+ 1 2)` to see if it works.

Installing Quicklisp

Quicklisp is a package manager for lisp. It handles downloading and installation of libraries. Installing it is rather easy. More information and documentation can be found at http://www.quicklisp.org/beta/

Download the file http://beta.quicklisp.org/quicklisp.lisp

Load it with sbcl or ccl:

```
$ sbcl --load quicklisp.lisp
```

This will load the file into lisp and we can proceed to install it. Type the following into the lisp prompt:

[1]http://ccl.clozure.com/manual/chapter2.html

```
(quicklisp-quickstart:install)
```

This will install quicklisp in your home directory.

In order to make sure quicklisp is loaded every time you start lisp, type the following:

```
(ql:add-to-init-file)
```

And you're done. You can now quickload libraries, for instance the following command will install the Restas web framework:

```
(ql:quickload "restas")
```

Recommended editors

- Emacs and Slime: The best option if you already know it, or you are willing to learn it.
- Vim and Slimv: The next best thing. Vim isn't actually easier to learn than Emacs, but if you already know it, it can get the job done.
- All the other options pretty much stink, but Kate at least has a built in terminal, so it's a bit easier to work with lisp interactively.

Windows

Getting a Lisp implementation

The implementation I recommend on Windows is CCL, you can download it from here[2].

After you've downloaded the file, uncompress it in the directory `C:\ccl`.

The `ccl` folder will have two executables, one named `wx86cl` for 32 bit systems, and `wx86cl64` for 64 bin systems.

At the command prompt, we can start the application by typing:

```
 > c:\ccl\wx86cl
```

Let's make it possible to start ccl simply by typing `ccl`. I'll demonstrate for the 32 bit version, it is equivalent for the 64 bit.

First, rename the `wx86cl` and `wx86cl.image` files to `ccl` and `ccl.image` respectively. Now, we need to set up the PATH enviromental variable so that windows knows where to find CCL.

For Windows 7, click the Start menu, and right click on `Computer` and select `properties`. From the sidebar select `Advanced system settings`. At the bottom of the window, click on the `Environment Variables` button. In the second pane, called `System variables`, search for the `Path` variable, select it, click on `Edit`. At The end of the `Variable value` field, append the following: `;C\ccl\`. Click OK. Open a command prompt, and type ccl, it should greet you with a message. That's it, you have CCL installed.

[2]http://ccl.clozure.com/download.html

Installing Quicklisp

Quicklisp is a package manager for lisp. It handles downloading and installation of libraries. Installing it is rather easy. More information and documentation can be found at http://www.quicklisp.org/beta/

Download the file http://beta.quicklisp.org/quicklisp.lisp

Open a command prompt, and go to the directory where you downloaded it:

```
 > chdir path\to\download\directory
```

Load it with ccl:

```
 > ccl --load quicklisp.lisp
```

This will load the file into lisp and we can proceed to install it. Type the following into the lisp prompt:

```
(quicklisp-quickstart:install)
```

This will install quicklisp in your home directory.

In order to make sure quicklisp is loaded every time you start lisp, type the following:

```
(ql:add-to-init-file)
```

You can now install lisp libraries using the `ql:quickload` command. Note that some libraries we'll be using depend on haveing OpenSSL installed, so make sure you install it, a third party installer is available from here[3]

Restart CCL, to test if it worked:

```
 ? (quit)
 > ccl
 ? (ql:quickload "restas")
```

If it started downloading and installing Restas, you're done. You can now quickload libraries.

Recommended editors

- Emacs and Slime: The best option if you already know it, or you are willing to learn it.
- Lisp Cabinet: A bundle of Emacs and various lisp implementations, an easy way to install Lisp and Emacs, with various customizations.
- Vim and Slimv: The next best thing. Vim isn't actually easier to learn than Emacs, but if you already know it, it can get the job done.
- Sublime Text2: Seems to be acceptable for editing lisp code.
- LispIDE: Barely qualifies as an IDE, but is an option you can look into.
- Notepad++: Popular code editor for Windows. Minimally acceptable as a lisp editor.

[3]http://slproweb.com/products/Win32OpenSSL.html

Appendix B: Recomended reading

Online tutorials

A lisp tutorial I like is Lisp in small parts[4]

If you are new to programming, people usually recommend the free book Common Lisp: A Gentle Introduction to Symbolic Computation[5].

For experienced hackers new to lisp, Practical Common Lisp[6] is probably the best way to learn the language.

Cliki: The Common Lisp wiki

Almost all information you would want about Common Lisp can be found on Cliki[7].

Cliki pages of note: Getting started[8]

Online tutorials[9]

Recomended libraries[10]

IRC

I(and many lispers) hang out on irc on the Freenode[11] server. Channels I frequent include `#lispweb` and `#lisp`. You can also find help on `#clnoobs`.

Check out a bunch of other lisp-related channels on cliki[12].

[4]http://lisp.plasticki.com/
[5]http://www-2.cs.cmu.edu/~dst/LispBook/
[6]http://www.gigamonkeys.com/book/
[7]http://cliki.net
[8]http://www.cliki.net/Getting%20Started
[9]http://www.cliki.net/Online%20Tutorial
[10]http://www.cliki.net/Current%20recommended%20libraries
[11]http://freenode.net/
[12]http://www.cliki.net/IRC