

AllegroCache with Web Servers

v1.1

Introduction

Using AllegroCache with a web server poses certain challenges that we'll describe in this document. The primary challenge is using AllegroCache in a multi-threaded environment. In a multi-threaded environment you need to worry about serializing operations on the database and handling conflicts should they arise. While we discuss these problems in the context of a web server what is said also applies to other multi-threaded applications.

The web server we'll be using for sample code is AllegroServe.

Multi-threading

In Allegro Common Lisp a program can create multiple threads of execution using such lisp functions as **mp:process-run-function**. In ACL threads are referred to as *processes* for historic reasons but we'll use the term thread since it better describes what they are.

In AllegroCache a program opens a connection to a database and that connection can only be used by one thread at a time. In a multi-threaded application such as a web server it's likely that many threads will need to access the AllegroCache database. This document will show some common ways to share a database connection or connections.

Password Database

Consider the following scenario. Your web site has pages that only authorized clients can view. You'll require users to supply a name and password to access the protected pages. Therefore you would like to keep a simple name/password database in AllegroCache. Furthermore this database can be updated by users wishing to change their passwords.

This example only shows a fraction of the power of AllegroCache. That is intentional. We aren't trying to demonstrate AllegroCache as much as we are demonstrating how to construct an application that uses AllegroCache. Once you know how the framework is built you can add in the details of your own application as needed.

What follows is one of the simplest solutions to this password database design. For more serious web sites where authorization is required we would recommend using AllegroServe's WebActions framework as this provides a better way to do authorization using sessions.

For our simple password database we don't need to define any classes. We can use the built-in **ac-map** class from AllegroCache. An object of this class will map a user name to that user's password.

Database code

We'll begin the code with a package definition for our sample code

```
(require :aserve)
(require :acache)

(defpackage :sample (:use :common-lisp :excl
                          :db.allegrocache
                          :net.aserve
                          :net.html.generator
                          ))
(in-package :sample)
```

Next we'll define some global variables.

```
(defparameter *password-db-file* "password.db")
(defvar *password-db*)
(defvar *password-map*)
(defvar *db-lock* (mp:make-process-lock))
```

The following function will create the database we'll be using. We only call this function once when we're setting up our web site. It allows us to set the main site password (which we'll need to use to add new users and do other administrative actions).

```
(defun create-password-database (db-file admin-name admin-password)
  (let (*allegrocache*)
    (create-file-database db-file)
    (unwind-protect
      (let ((map (make-instance 'ac-map :ac-map-name "password")))
        (setf (map-value map admin-name) admin-password)
        (commit))
      (close-database))))
```

You'll note that we create a new binding for the special variable `*allegrocache*`. This isn't strictly necessary but is good style. We know that `create-file-database` will set the value of `*allegrocache*` and by binding it here we prevent that setting of `*allegrocache*` from affecting other threads running in the same Lisp which may be making use of the global value of `*allegrocache*`.

After creating the file database we make the map instance we'll be using for our database and we give that map instance the name "password." Then we commit the creation of the map and close the database. Now the database is ready to be used by our application.

We are using AllegroCache in *standalone mode*. For our simple web site we will have only one web server, thus we won't need to create a shared name/password database.

Each time the web server starts it will have to open up this database and prepare to use it. This function does that work:

```
(defun open-password-database (db-file)
  (let (*allegrocache*)
    (open-file-database db-file)
    (setf *password-db* *allegrocache*))
```

```
(setf *password-map*
      (retrieve-from-index 'ac-map 'ac-map-name "password"))))
```

Again we create a private binding of `*allegrocache*` so that the call to `open-file-database` doesn't change the global value of `*allegrocache*`. Next we store the database connection object in the global variable `*password-db*` and we store a pointer to the `ac-map` object we'll need in the variable `*password-map*`. We do not close the database.

At some point we'll need to look up a password given a user name. This function will do that:

```
(defun get-password-for (user)
  (mp:with-process-lock (*db-lock*)
    (let ((*allegrocache* *password-db*))
      (map-value *password-map* user))))
```

The call to `map-value` looks up the password for the given name. Before we can call `map-value` we have to make sure it's safe to do so. In the standalone AllegroCache we aren't concerned about multiple threads calling `map-value` at the same time. (That would however be a problem in the client/server AllegroCache). What we're worried about is one thread doing a commit and another calling `map-value` on the same database connection. That can cause problems. Thus to be safe we have a global lock `*db-lock*` which we must grab before doing any database operations. Given that the `map-value` call is very fast it's not a burden to keep the database locked away from other threads during the execution of this call. We locally bind `*allegrocache*` to the database connection we'll be using. Many AllegroCache functions implicitly use the value of `*allegrocache*` so it's critical that it be bound to the current connection when AllegroCache functions such as `map-value` are called.

`*password-map*` has an `ac-map` object as its value. This value can only be examined when `*allegrocache*` is bound to the database connection from which the `ac-map` object was obtained.

We've taken some pains to keep from modifying the global value of `*allegrocache*` in order to demonstrate a good style of programming. When you're debugging your application and need to view objects interactively you may wish to set the global value `*allegrocache*` to the active database connection, thus allowing object inspectors to work.

Setting a password value is very similar to getting the value:

```
(defun set-password-for (user new-password)
  (mp:with-process-lock (*db-lock*)
    (let ((*allegrocache* *password-db*))
      (setf (map-value *password-map* user) new-password)
      (commit)))))
```

One major difference is the call to `commit`. Because we're using standalone AllegroCache the `commit` cannot fail like it can in client/server mode. Thus we don't need any code to retry the operation if the `commit` fails.

And finally it would be nice to have a function to close the database.

```
(defun close-password-database ()
  (close-database :db *password-db*))
```

It's a good idea to close the database explicitly before exiting the application. This ensures that all partially filled buffers are flushed to the disk. In this simple case the close-database isn't necessary since the only changes that are made are immediately followed by a commit which flushes all written buffers to the disk. Thus there should be no work for close-database to do. However calling close-database is always good style.

Testing the database code

We can test out our database code before connecting it to the web server.

First we need to load in AllegroServe and AllegroCache.

```
cl-user(1): (require :aserve)
; Fast loading /usr/acl/aserve.fasl
t
cl-user(2): (require :acache)
; Fast loading /usr/acl/acache.fasl
AllegroCache version 1.0.1
t
```

Next we load in the code shown above and switch to the package in which we've placed this code

```
cl-user(4): :cl password
; Fast loading /usr/tmp/password.fasl
cl-user(5): :package sample
sample(6):
```

We build the password database, put in the name/password pair admin/foo and close the database:

```
sample(6): (create-password-database *password-db-file* "admin" "foo")
t
sample(7):
```

With the database built we can now open it up and perform some operations:

```
sample(7): (open-password-database *password-db-file*)
#<ac-map oid: 12, ver 5, trans: 7, not modified @ #x71dee9d2>
sample(8): (get-password-for "admin")
"foo"
t
sample(11): (set-password-for "smith" "john")
t
sample(12): (get-password-for "smith")
"john"
t
sample(13): (set-password-for "smith" "fred")
t
```

```
sample(14): (get-password-for "smith")
"fred"
t
sample(15):
```

The AllegroServe code

Now that we've written and tested the database code for our password database let's look at the AllegroServe code to make use of it.

We start the server with

```
(defun start-test (&key (port 9999))
  (open-password-database *password-db-file*)
  (start :port port)
  )
```

The open-password-database function is given above.

Let's first show how to use the password database in the most transparent way possible. Here is how we would publish a page which needed the user to supply the correct name/password values before it could be viewed:

```
(publish
:path  "/secret-page"
:content-type "text/html"
:function
  #'(lambda (req ent)
    (multiple-value-bind (name password)
      (get-basic-authorization req)
      (if* (and (> (length name) 0)
        (equal password
          (get-password-for name)))
        then ; success!!
          (with-http-response (req ent)
            (with-http-body (req ent)
              (html (:head (:title "Secret Page"))
                (:body "Here is the secret word: fish"))))
          else ; need correct name/password
            (with-http-response (req ent)
              :response *response-unauthorized*
              (set-basic-authorization req "secretserver")
              (with-http-body (req ent)))))))
```

We make use of the get-password-for function we defined above to see if the the user has supplied the correct name/password values.

If you have only one page to protect then the above solution is acceptable, but if you wish to protect a whole group of pages then it's better to use an authorizer object. You would begin by building such an object:

```
(defvar *ac-authorizer*
  (make-instance 'function-authorizer
    :function
    #'(lambda (req ent auth)
      (declare (ignore auth))
      (multiple-value-bind (name password)
        (get-basic-authorization req)
        (if* (and (> (length name) 0)
              (equal password
                    (get-password-for name)))
          then t ; authorized
          else ; send back request for name/password
            (with-http-response (req ent
                                :response *response-
                                unauthorized*)
              (set-basic-authorization req "secretserver")
              (with-http-body (req ent)))
            :done))))))
```

and then you can use this object with an entity you wish to protect:

```
(publish
  :path "/other-secret-page"
  :content-type "text/html"
  :authorizer *ac-authorizer*
  :function #'(lambda (req ent)
    (with-http-response (req ent)
      (with-http-body (req ent)
        (html (:head (:title "Secret Page"))
              (:body "Here is the secret word:
cake"))))))))
```

The following page allows the user to change their password:

```
(publish
  :path "/change-password"
  :content-type "text/html"
  :function
  #'(lambda (req ent)
    (if* (eq (request-method req) :post)
      then ; do password change if authorized
        (let ((name (request-query-value "name" req))
              (old (request-query-value "old" req))
              (new (request-query-value "new" req)))
          (if* (and (> (length name) 0)
```

```

        (> (length new) 0)
        (equal old (get-password-for name)))
    then (set-password-for name new)
        (with-http-response (req ent)
            (with-http-body (req ent)
                (html (:body "Password change successful"))))
    else (with-http-response (req ent)
        (with-http-body (req ent)
            (html (:body "Password change unsuccessful"
                :br
                ((:a :href "/change-password")
                    "Retry"))))))))
else ; put up form for password change
    (with-http-response (req ent)
        (with-http-body (req ent)
            (html (:h1 "Password Change")
                ( (:form :method "POST")
                    "Name: " ((:input :type "text"
                        :name "name"))
                    :br
                    "Current Password: "
                    ((:input :type "password"
                        :name "old"))
                    :br
                    "New Password: "
                    ((:input :type "password"
                        :name "new"))
                    :br
                    ((:input :type "submit"))))))))

```

Database in client/server mode

For this application we chose to use AllegroCache in standalone mode. This is the appropriate mode for a simple single-machine web server. It is instructive however to consider what it would take to convert this example to use AllegroCache in client/server mode. A password database built this way will allow multiple web servers to access the database and would also allow one to write administrative programs to add accounts to the database and generate reports based on the contents of the database.

The database creation function will start a server and then create a connection to the server. A server needs to run on a particular port so we must specify that.

```

(defparameter *server-port* 8888)

(defun create-password-database-cs (db-file admin-name admin-password)
  (let (*allegrocache*)
    (start-server db-file *server-port*
      :if-exists :supersede
      :if-does-not-exist :create)
    (open-network-database "localhost" *server-port*)

```



```
(unwind-protect
  (let ((map (make-instance 'ac-map :ac-map-name "password"))))
    (setf (map-value map admin-name) admin-password)
    (commit))
  (close-database :stop-server t))))
```

When we call `close-database` we specify that the server should shut down as well.

In order to connect to the database we start up the server on the database and then create a connection to the database. If we wanted to have multiple web servers connect to this database then we would separately start the server and the `open-password-database-cs` function would only open-network-database.

```
(defun open-password-database-cs (db-file)
  (let (*allegrocache*)
    (start-server db-file *server-port*)
    (open-network-database "localhost" *server-port*)
    (setf *password-db* *allegrocache*)
    (setf *password-map*
      (retrieve-from-index 'ac-map 'ac-map-name "password"))))
```

There is a subtle but important change in the password retrieval function, namely the addition of a call to `rollback`.

```
(defun get-password-for-cs (user)
  (mp:with-process-lock (*db-lock*)
    (let ((*allegrocache* *password-db*))
      (rollback)
      (map-value *password-map* user))))
```

We first do a `rollback` to advance this connection's view of the database to the most recent time. A `rollback` operation does two things: it undoes any changes we've made (and we've made none so this has no effect) and it advances the view of the database so we're seeing the result of the most recent commit. Since at this point we only have a single client connection to the database this call to `rollback` is unnecessary, but it's good style and it means we're ready should we add more connections to the database.

The only other change from our standalone version is to the function to set the password

```
(defun set-password-for-cs (user new-password)
  (mp:with-process-lock (*db-lock*)
    (let ((*allegrocache* *password-db*))
      (with-transaction-restart ()
        (setf (map-value *password-map* user) new-password)
        (commit))))))
```

We've added the `with-transaction-restart` macro so that the database modification and commit is retried if the commit fails. If there's only one client connection then the commit can never fail so until we

actually start sharing the database with multiple web servers the use of this macro is unnecessary (but it is good style).

Connection pools

So far we've had one connection to the database that all threads shared. For the password database that's appropriate. But let's imagine that each thread is doing more than just looking up a password. The thread may be navigating through the persistent object space or creating new objects and committing them. In that case we don't want the bottleneck of a single database connection. We would like each thread to have its own connection to the database. In practice though we know that each database connection takes up system resources so we don't want to create database connections that we don't really need. Thus we introduce the idea of a *connection pool*, which is a set of database connections that all threads share. If a thread needs to do a database operation it takes a connection out of the pool, uses it and then puts it back when its done. If there are no connections in the pool a thread can either create a new connection or it can wait for an existing connection to be freed by another thread.

We'll show how a connection pool can be created for our simple password database. We acknowledge that for this simple example there is little need for a connection pool however you should imagine that each thread is doing a lot more than just looking up passwords when it gets a database connection.

The connection pool is managed by the `with-client-connection` macro we'll show below. This macro retrieves a connection from the pool. If no connection is available it will create a new connection but it will create no more than `*connection-count*` connections. If all connections are created and a new connection is needed the requesting thread will be put to sleep until a connection is returned to the pool.

In our password example we'll need to access the map object that contains the password data. In the preceding examples we've stored the map object in the global variable `*password-map*`. This strategy will **not** work in our connection pool case since each database connection will have it's own unique object that represents the single map object in the database. Thus we need to store a map object for each connection. Our connection pool will hold the database object and either the map object or nil. The code will look up and cache the map object as needed. That makes this connection pool code application specific. You'll likely want to tune your connection pool code for your application as well.

The global variables we'll need are the following. `*client-connections*` will be a list of poolobj objects. The lock `*pool-lock*` must be held before a thread can look at `*client-connections*`. The gate `*pool-gate*` will be used to block threads waiting for a connection to be freed. `*connection-host*` and `*connection-port*` describe the location of the database server. `*connection-count*` is the number of database connections yet to be allocated.

```
(defvar *client-connections* nil)

(defvar *pool-lock* (mp:make-process-lock))
(defvar *pool-gate* (mp:make-gate t))

(defvar *connection-host*)
```

```
(defvar *connection-port*)  
(defvar *connection-count*)
```

The connection pool consists of a list of poolobj objects. The database slot contains the database connection and the map slot is used to cache a pointer to the ac-map object named “password”.

```
(defstruct poolobj  
  database  
  map)
```

```
(defmacro with-client-connection (var &body body)
  (let ((alloc-connection (gensym)))
    `(let (,var ,alloc-connection)

      (loop
        (mp:with-process-lock (*pool-lock*)
          (if* (null (setq ,var (pop *client-connections*)))
            then (if* (> *connection-count* 0)
              then (decf *connection-count*)
              (setq ,alloc-connection t)
              else (mp:close-gate *pool-gate*))))

        (if* ,var
          then (return)
          elseif ,alloc-connection
          then (let (*allegrocache*)
              (setq ,var (make-poolobj
                :database (open-network-database
                  *connection-host*
                  *connection-port*)))
              (return))
          else (mp:process-wait "gate" #'mp:gate-open-p *pool-gate*)))

      (unwind-protect
        (let ((*allegrocache* (poolobj-database ,var)))
          ,@body)
        (mp:with-process-lock (*pool-lock*)
          (push ,var *client-connections*)
          (mp:open-gate *pool-gate*))))))
```

In `with-client-connection` we first see if a connection is available and if so it is returned. If not then we see if there are any connections yet to be allocated. If so we allocate the connection. If not we block on the gate `*pool-gate*`. If a connection is available we bind `*allegrocache*` to the database object and run the body of the macro. Once the body finishes we put the connection back in the pool and open the gate so that any threads waiting for a connection are awoken.

To use this code we must start the server just once

```
(defun start-password-database-server (db-file)
  (start-server db-file *server-port*))
```

Then when the web server starts up we call this function to initialize all the global variables

```
(defun open-password-database-pool ()
  (setq *connection-host* "localhost"
        *connection-port* *server-port*
        *connection-count* 5
        *client-connections* nil))
```

The code to retrieve a password may have to first find the map object and cache it before it can lookup the user in the map.

```
(defun get-password-for-pool (user)
  (with-client-connection poolobj
    (rollback)
    (map-value (or (poolobj-map poolobj)
                  (setf (poolobj-map poolobj)
                        (retrieve-from-index 'ac-map
                                           'ac-map-name "password"))))
              user)))
```

Note that we're no longer grabbing the `*db-lock*` as we did in preceding examples. That is because there is no longer a global lock to access the database. Instead each thread runs independently and AllegroCache is dealing with all concurrency issues.

Setting the password is written in a similar way. Since we're doing a commit we have to use `with-transaction-restart`.

```
(defun set-password-for-pool (user new-password)
  (with-client-connection poolobj
    (with-transaction-restart nil
      (setf (map-value (or (poolobj-map poolobj)
                          (setf (poolobj-map poolobj)
                                (retrieve-from-index 'ac-map
                                                    'ac-map-name
                                                    "password"))))
            user)
            new-password)
    (commit))))
```

Webactions with AllegroCache

Sophisticated web sites should be written using a higher level web programming framework such as Webactions. In this section we'll discuss how to integrate AllegroCache into a Webactions project.

One of the key features of Webactions is its session support. A session is information associated with a particular web browser that lasts through many http requests. Webactions stores the session information in the web server's memory.

In our first scenario we'll take a case of a web site that has multiple web servers serving the same pages. The user just asks for <http://www.this-popular-site.com> and that request is sent to one of the N web servers serving this site due to some IP redirection software at the server end. Webactions sessions are inadequate in this scenario since session data stored in one web server will not be available to the other N-1 web servers. What we need is a common database that all web servers can use to store the session data. This is where AllegroCache is very useful.

First let's review how Webactions stores sessions. Each session has a unique session key which is a long string. When a web browser contacts a server it passes its session key (if it has one) to that server. If no session key is passed then the server creates a new session and sends that session's key back to the web browser along with the response. Thus after one request/response the server and web browser have established a session and agreed on its key.

If a web browser passes a session key to the server that the server does not recognize, the server creates a session with that key. This means that if you have multiple servers for a site then after the first server creates a session and passes back the key if the web browser should happen to send a request to a different web server that web server will create a session with the same key. This is important since it's the session key that will be used to find the persistent object that's associated with the session in the AllegroCache database.

For example, suppose a user starts his web browser and visits <http://www.mysite.com> which is implemented using Webactions. The web server will find that no session key was sent with the request so it will create a new session and with its reply send back the session key, which we'll say is "abcd1234". On the next request from the same browser to a page on <http://www.mysite.com> the browser will send the session key "abcd1234". If this request is sent to the same web server as handled the original request then it will cause that web server to locate in its table of sessions the session named "abcd1234" and pass that session object to the code handling the request. If that request goes to another web server however then that web server will not find a session named "abcd1234". It will then create a session named "abcd1234" and pass that session object to the code handling the request. These session objects are transient values stored in the Lisp heap of each process. They have no relation to each other. What we show in this example is how to use AllegroCache to allow those session objects to share memory so that values set in one session can be read in another.

Persistent Session Example

We'll show how two distinct web servers can share session data. The web servers can be running on different machines or the same machine (although to make our example as simple as possible we use the machine name "localhost" so both lisp servers and the database server must run on the same machine.)

In our example we have two pieces of data we store in our session, one called 'a' and the other called 'b'. We create a Webactions project which has one page that shows the current values of 'a' and 'b' and allows the user to change either value. That value, once changed, will be visible to all web servers running this sample code.

The complete lisp code for this is this file:

```
;; persistent sessions
(eval-when (compile load eval)
  (require :aserve)
  (require :webactions))
```

```

(require :acache))

(defpackage :wasample (:use :common-lisp :excl
                           :db.allegrocache
                           :net.aserve
                           :net.html.generator
                           ))

(in-package :wasample)

(webaction-project "wasample"
                  :project-prefix "/"
                  :destination "" ; current directory
                  :index "main"
                  :map '(("main" action-get-session-vals "main.clp")
                        ("setvalues" action-get-session-vals
                                   action-process-form
                                   action-store-session-vals
                                   "main" (:redirect t))))

(defclass psession ()
  ((key :initarg :key :index :any-unique
        :reader psession-key)

   ;; sample user data
   (a :initform nil
      :accessor psession-a)
   (b :initform nil
      :accessor psession-b))

  (:metaclass persistent-class))

(defvar *pool-lock* (mp:make-process-lock))
(defvar *pool-gate* (mp:make-gate t))

(defvar *client-connections* nil)
(defvar *connection-host*)
(defvar *connection-port*)
(defvar *connection-count*)

(defmacro with-client-connection (ignore &body body)
  (declare (ignore ignore))

  (let ((alloc-connection (gensym))
        (var (gensym)))
    `(let (,var ,alloc-connection)

       (loop
        (mp:with-process-lock (*pool-lock*)
          (if* (null (setq ,var (pop *client-connections*)))
              then (if* (> *connection-count* 0)
                      then (decf *connection-count*)
                          (setq ,alloc-connection t)
                      else (mp:close-gate *pool-gate*))))

        (if* ,var
            then (return)
            elseif ,alloc-connection
            then (setq ,var (open-network-database
                          *connection-host*

```

```

        *connection-port*
        :use :memory))
    (return)
    else (mp:process-wait "gate" #'mp:gate-open-p *pool-gate*)))

(unwind-protect
  (let ((*allegrocache* ,var))
    ,@body)
  (mp:with-process-lock (*pool-lock*)
    (push ,var *client-connections*)
    (mp:open-gate *pool-gate*))))))

(defmacro get-psession-object (key modified)
  `(or (retrieve-from-index 'psession 'key ,key)
    (progn , (if* modified
      then `(setq ,modified t))
      (make-instance 'psession :key ,key))))

(defun action-get-session-vals (req ent)
  (declare (ignore ent))
  (let* ((session (web-session-from-req req))
    (key (web-session-key session))
    (modified))
    (with-client-connection ()
      (rollback)
      (with-transaction-restart ()
        (let ((psession (get-psession-object key modified)))

          (setf (web-session-variable session "a")
            (psession-a psession))

          (setf (web-session-variable session "b")
            (psession-b psession))
          (if* modified then (commit))))))

    :continue)

(defun action-process-form (req ent)
  (declare (ignore ent))
  ;; process the values stored in the form
  (let ((set-a (request-query-value "set-a" req))
    (set-b (request-query-value "set-b" req))
    (session (web-session-from-req req))
    )

    (if* set-a
      then (setf (web-session-variable session "a") set-a))

    (if* set-b
      then (setf (web-session-variable session "b") set-b))

    :continue))

(defun action-store-session-vals (req ent)
  (declare (ignore ent))
  (let* ((session (web-session-from-req req))

```



```

        (key (web-session-key session)))
(with-client-connection ()
  (rollback)
  (with-transaction-restart ()
    (let ((p-session (get-p-session-object key nil)))

      (setf (p-session-a p-session)
            (web-session-variable session "a"))
      )

      (setf
        (p-session-b p-session)
        (web-session-variable session "b"))
      )
    (commit)))

:continue
)

(defun start-db-server (port &optional create)
  (start-server "wasample.db" port
    :if-exists (if* create then :supersede else :open)
    :if-does-not-exist
    (if* create then :create else :error)))

(defun start-webserver (web-port db-port)
  (setq *connection-host* "localhost"
        *connection-port* db-port
        *connection-count* 5)

  (net.aserve:start :port web-port))

```

There is one file `main.clp` that's needed as well

```

<html>
<head><title>Webactions/AllegroCache Session test</title></head>
<body>
The value of A is <clp_value name="a" session/>.
<br>
<form action="setvalues">
Set A to <input type="text" name="set-a">
<input type="submit" value="Set A">
</form>
<hr>
The value of B is <clp_value name="b" session/>.
<br>
<form action="setvalues">
Set B to <input type="text" name="set-b">
<input type="submit" value="Set B">
</form>
</body>
</html>

```

In order to run this example you need to choose a port for the AllegroCache server to listen on and then a port for each web server. If you want AllegroCache to listen on port 8888 and then to start up web servers on ports 8000, 8001 etc you would run the example by executing the following commands:

```
(wasample::start-db-server 8888 t) ; t means create fresh database
```

Then on each process you want to be a web server do

```
(wasample::start-webserver 8000 8888)
```

and then on the next process

```
(wasample::start-webserver 8001 8888)
```

and so on. Start a browser on the machine and visit <http://localhost:8000> and then start a different browser and visit <http://localhost:8001>. Set and read the results on different browsers and convince yourself that the session data is shared between the two web servers.

Now let's go back and look at the code. We're using the connection pooling idea introduced before. This time however our pool is simply a list of connections and not a structure holding a connection. We do this because this is what's appropriate for this example. The `get-psession-object` is responsible for returning the persistent object that's associated with a session. Note that it tries to get the object first before creating one. There is a possibility that two lisp threads in the same web server will try to create a `psession` object with the same key. This would be a disaster if it occurred but it can't occur since we've declared the key slot of the `psession` object to have a unique index. Thus the database connection that tried to commit a `psession` object with a given key will get a commit error. The `with-transaction-restart` macro will then cause the transaction to be rolled back and `get-psession-object` will be called again. This time the `retrieve-from-index` function will find that existing `psession` object rather than creating a new one. So the next time the commit is done it won't fail.

In your code where you use unique indexes you'll want to use the strategy shown in `get-psession-object` to retrieve or create an object.

You'll note that our strategy for using AllegroCache with Webactions is to do quick database operations and not to keep a database connection during a whole sequence of actions. This is certainly the simplest solution and a safe one. If you just allocate a database connection from the pool and then do a sequence of actions then you may be in trouble if the commit fails since you then have to do the rollback and repeat the actions yourself. In our code the `action-store-session-vals` modifies the database and the commit here can fail. However its easy to see that if the commit fails its easy for the connection to be rolled back and the store operations repeated.