29. Practical: An MP3 Browser

The final step in building the MP3 streaming application is to provide a Web interface that allows a user to find the songs they want to listen to and add them to a playlist that the Shoutcast server will draw upon when the user's MP3 client requests the stream URL. For this component of the application, you'll pull together several bits of code from the previous few chapters: the MP3 database, the `define-url-function` macro from Chapter 26, and, of course, the Shoutcast server itself.

Playlists

The basic idea behind the interface will be that each MP3 client that connects to the Shoutcast server gets its own *playlist*, which serves as the source of songs for the Shoutcast server. A playlist will also provide facilities beyond those needed by the Shoutcast server: through the Web interface the user will be able to add songs to the playlist, delete songs already in the playlist, and reorder the playlist by sorting and shuffling.

You can define a class to represent playlists like this:

```
(defclass playlist ()
  ((id           :accessor id           :initarg :id)
   (songs-table  :accessor songs-table  :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx  :accessor current-idx  :initform 0)
   (ordering     :accessor ordering     :initform :album)
   (shuffle      :accessor shuffle      :initform :none)
   (repeat       :accessor repeat       :initform :none)
   (user-agent   :accessor user-agent   :initform "Unknown")
   (lock         :reader   lock         :initform (make-process-lock))))
```

The `id` of a playlist is the key you extract from the request object passed to `find-song-source` when looking up a playlist. You don't actually need to store it in the `playlist` object, but it makes debugging a bit easier if you can find out from an arbitrary playlist object what its `id` is.

The heart of the playlist is the `songs-table` slot, which will hold a `table` object. The schema for this table will be the same as for the main MP3 database. The function `make-playlist-table`, which you use to initialize `songs-table`, is simply this:

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

The Package

You can define the package for the code in this chapter with the following **DEFPACKAGE**:

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

Because this is a high-level application, it uses a lot of lower-level packages. It also imports three symbols from the `ACL-SOCKET` package and two more from `MULTIPROCESSING` since it just needs those five and not the other 139 symbols those two packages export.

By storing the list of songs as a table, you can use the database functions from Chapter 27 to manipulate the playlist: you can add to the playlist with `insert-row`, delete songs with `delete-rows`, and reorder the playlist with `sort-rows` and `shuffle-table`.

The `current-song` and `current-idx` slots keep track of which song is playing: `current-song` is an actual `song` object, while `current-idx` is the index into the `songs-table` of the row representing the current song. You'll see in the section "Manipulating the Playlist" how to make sure `current-song` is updated whenever `current-idx` changes.

The `ordering` and `shuffle` slots hold information about how the songs in `songs-table` are to be ordered. The `ordering` slot holds a keyword that tells how the `songs-table` should be sorted when it's not shuffled. The legal values are `:genre`, `:artist`, `:album`, and `:song`. The `shuffle` slot holds one of the keywords `:none`, `:song`, or `:album`, which specifies how `songs-table` should be shuffled, if at all.

The `repeat` slot also holds a keyword, one of `:none`, `:song`, or `:all`, which specifies the repeat mode for the playlist. If `repeat` is `:none`, after the last song in the `songs-table` has been played, the `current-song` goes back to a default MP3. When `:repeat` is `:song`, the playlist keeps returning the same `current-song` forever. And if it's `:all`, after the last song, `current-song` goes back to the first song.

The `user-agent` slot holds the value of the User-Agent header sent by the MP3 client in its request for the stream. You need to hold onto this value purely for use in the Web interface--the User-Agent header identifies the program that made the request, so you can display the value on the page that lists all the playlists to make it easier to tell which playlist goes with which connection when multiple clients connect.

Finally, the `lock` slot holds a *process lock* created with the function `make-process-lock`, which is part of Allegro's `MULTIPROCESSING` package. You'll need to use that lock in certain functions that manipulate `playlist` objects to ensure that only one thread at a time manipulates a given playlist object. You can define the following macro, built upon the

with-process-lock macro from MULTIPROCESSING, to give an easy way to wrap a body of code that should be performed while holding a playlist's lock:

```
(defmacro with-playlist-locked ((playlist) &body body)
  `(with-process-lock ((lock ,playlist))
     ,@body))
```

The with-process-lock macro acquires exclusive access to the process lock given and then executes the body forms, releasing the lock afterward. By default, with-process-lock allows recursive locks, meaning the same thread can safely acquire the same lock multiple times.

Playlists As Song Sources

To use playlists as a source of songs for the Shoutcast server, you'll need to implement a method on the generic function find-song-source from Chapter 28. Since you're going to have multiple playlists, you need a way to find the right one for each client that connects to the server. The mapping part is easy--you can define a variable that holds an **EQUAL** hash table that you can use to map from some identifier to the playlist object.

```
(defvar *playlists* (make-hash-table :test #'equal))
```

You'll also need to define a process lock to protect access to this hash table like this:

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

Then define a function that looks up a playlist given an ID, creating a new playlist object if necessary and using with-process-lock to ensure that only one thread at a time manipulates the hash table.[1]

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

Then you can implement find-song-source on top of that function and another, playlist-id, that takes an AllegroServe request object and returns the appropriate playlist identifier. The find-song-source function is also where you grab the User-Agent string out of the request object and stash it in the playlist object.

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent)))
        (when user-agent (setf (user-agent playlist) user-agent))))
    playlist))
```

The trick, then, is how you implement playlist-id, the function that extracts the identifier from the request object. You have a couple options, each with different implications for the user interface. You can pull whatever information you want out of the request object, but however you decide to identify the client, you need some way for the user of the Web interface to get hooked up to the right playlist.

For now you can take an approach that "just works" as long as there's only one MP3 client per machine connecting to the server and as long as the user is browsing the Web interface from the machine running the MP3 client: you'll use the IP address of the client machine as the identifier. This way you can find the right playlist for a request regardless of whether the request is from the MP3 client or a Web browser. You will, however, provide a way in the Web interface to select a different playlist from the browser, so the only real constraint this choice puts on the application is that there can be only one connected MP3 client per client IP address.[2] The implementation of `playlist-id` looks like this:

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

The function `request-socket` is part of AllegroServe, while `remote-host` and `ipaddr-to-dotted` are part of Allegro's socket library.

To make a playlist usable as a song source by the Shoutcast server, you need to define methods on `current-song`, `still-current-p`, and `maybe-move-to-next-song` that specialize their `source` parameter on `playlist`. The `current-song` method is already taken care of: by defining the accessor `current-song` on the eponymous slot, you automatically got a `current-song` method specialized on `playlist` that returns the value of that slot. However, to make accesses to the `playlist` thread safe, you need to lock the `playlist` before accessing the `current-song` slot. In this case, the easiest way is to define an `:around` method like the following:

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

Implementing `still-current-p` is also quite simple, assuming you can be sure that `current-song` gets updated with a new `song` object only when the current song actually changes. Again, you need to acquire the process lock to ensure you get a consistent view of the `playlist`'s state.

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

The trick, then, is to make sure the `current-song` slot gets updated at the right times. However, the current song can change in a number of ways. The obvious one is when the Shoutcast server calls `maybe-move-to-next-song`. But it can also change when songs are added to the playlist, when the Shoutcast server has run out of songs, or even if the playlist's repeat mode is changed.

Rather than trying to write code specific to every situation to determine whether to update `current-song`, you can define a function, `update-current-if-necessary`, that updates `current-song` if the `song` object in `current-song` no longer matches the file that the `current-idx` slot says should be playing. Then, if you call this function after any manipulation of the playlist that could possibly put those two slots out of sync, you're sure to

keep `current-song` set properly. Here are `update-current-if-necessary` and its helper functions:

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                 (file-for-current-idx playlist))
    (reset-current-song playlist)))

(defun file-for-current-idx (playlist)
  (if (at-end-p playlist)
    nil
    (column-value (nth-row (current-idx playlist) (songs-table playlist)) :file)))

(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

You don't need to add locking to these functions since they'll be called only from functions that will take care of locking the playlist first.

The function `reset-current-song` introduces one more wrinkle: because you want the playlist to provide an endless stream of MP3s to the client, you don't want to ever set `current-song` to **`NIL`**. Instead, when a playlist runs out of songs to play--when `songs-table` is empty or after the last song has been played and `repeat` is set to `:none`--then you need to set `current-song` to a special song whose file is an MP3 of silence[3] and whose title explains why no music is playing. Here's some code to define two parameters, `*empty-playlist-song*` and `*end-of-playlist-song*`, each set to a song with the file named by `*silence-mp3*` as their file and an appropriate title:

```
(defparameter *silence-mp3* ...)

(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
   'song
   :file file
   :title title
   :id3-size (if (id3-p file) (size (read-id3 file)) 0)))

(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))

(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist."))
```

`reset-current-song` uses these parameters when the `current-idx` doesn't point to a row in `songs-table`. Otherwise, it sets `current-song` to a `song` object representing the current row.

```
(defun reset-current-song (playlist)
  (setf
   (current-song playlist)
   (cond
     ((empty-p playlist) *empty-playlist-song*)
     ((at-end-p playlist) *end-of-playlist-song*)
     (t (row->song (nth-row (current-idx playlist) (songs-table playlist)))))))

(defun row->song (song-db-entry)
  (with-column-values (file song artist album id3-size) song-db-entry
    (make-instance
     'song
     :file file
     :title (format nil "~a by ~a from ~a" song artist album)
     :id3-size id3-size)))
```

```
(defun empty-p (playlist)
  (zerop (table-size (songs-table playlist))))
```

Now, at last, you can implement the method on `maybe-move-to-next-song` that moves `current-idx` to its next value, based on the playlist's repeat mode, and then calls `update-current-if-necessary`. You don't change `current-idx` when it's already at the end of the playlist because you want it to keep its current value, so it'll point at the next song you add to the playlist. This function must lock the playlist before manipulating it since it's called by the Shoutcast server code, which doesn't do any locking.

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all  (setf (current-idx playlist)
                       (mod (1+ (current-idx playlist))
                            (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist))))
```

Manipulating the Playlist

The rest of the playlist code is functions used by the Web interface to manipulate `playlist` objects, including adding and deleting songs, sorting and shuffling, and setting the repeat mode. As in the helper functions in the previous section, you don't need to worry about locking in these functions because, as you'll see, the lock will be acquired in the Web interface function that calls these.

Adding and deleting is mostly a question of manipulating the `songs-table`. The only extra work you have to do is to keep the `current-song` and `current-idx` in sync. For instance, whenever the playlist is empty, its `current-idx` will be zero, and the `current-song` will be the `*empty-playlist-song*`. If you add a song to an empty playlist, then the index of zero is now in bounds, and you should change the `current-song` to the newly added song. By the same token, when you've played all the songs in a playlist and `current-song` is `*end-of-playlist-song*`, adding a song should cause `current-song` to be reset. All this really means, though, is that you need to call `update-current-if-necessary` at the appropriate points.

Adding songs to a playlist is a bit involved because of the way the Web interface communicates which songs to add. For reasons I'll discuss in the next section, the Web interface code can't just give you a simple set of criteria to use in selecting songs from the database. Instead, it gives you the name of a column and a list of values, and you're supposed to add all the songs from the main database where the given column has a value in the list of values. Thus, to add the right songs, you need to first build a table object containing the desired values, which you can then use with an `in` query against the song database. So, `add-songs` looks like this:

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                'table
                :schema (extract-schema (list column-name) (schema *mp3s*)))))
```

```
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist))))
  (update-current-if-necessary playlist))
```

Deleting songs is a bit simpler; you just need to be able to delete songs from the `songs-table` that match particular criteria--either a particular song or all songs in a particular genre, by a particular artist, or from a particular album. So, you can provide a `delete-songs` function that takes keyword/value pairs, which are used to construct a `matching :where` clause you can pass to the `delete-rows` database function.

Another complication that arises when deleting songs is that `current-idx` may need to change. Assuming the current song isn't one of the ones just deleted, you'd like it to remain the current song. But if songs before it in `songs-table` are deleted, it'll be in a different position in the table after the delete. So after a call to `delete-rows`, you need to look for the row containing the current song and reset `current-idx`. If the current song has itself been deleted, then, for lack of anything better to do, you can reset `current-idx` to zero. After updating `current-idx`, calling `update-current-if-necessary` will take care of updating `current-song`. And if `current-idx` changed but still points at the same song, `current-song` will be left alone.

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))

(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
         (matcher (matching table :file (file (current-song playlist))))
         (pos 0))
    (do-rows (row table)
      (when (funcall matcher row)
        (return-from position-of-current pos))
      (incf pos))))
```

You can also provide a function to completely clear the playlist, which uses `delete-all-rows` and doesn't have to worry about finding the current song since it has obviously been deleted. The call to `update-current-if-necessary` will take care of setting `current-song` to **`NIL`**.

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

Sorting and shuffling the playlist are related in that the playlist is always either sorted *or* shuffled. The `shuffle` slot says whether the playlist should be shuffled and if so how. If it's set to `:none`, then the playlist is ordered according to the value in the `ordering` slot. When `shuffle` is `:song`, the playlist will be randomly permuted. And when it's set to `:album`, the list of albums is randomly permuted, but the songs within each album are listed in track order. Thus, the `sort-playlist` function, which will be called by the Web interface code whenever the user selects a new ordering, needs to set `ordering` to the desired ordering and set

`shuffle` to `:none` before calling `order-playlist`, which actually does the sort. As in `delete-songs`, you need to use `position-of-current` to reset `current-idx` to the new location of the current song. However, this time you don't need to call `update-current-if-necessary` since you know the current song is still in the table.

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

In `order-playlist`, you can use the database function `sort-rows` to actually perform the sort, passing a list of columns to sort by based on the value of `ordering`.

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist)
    (case (ordering playlist)
      (:genre  '(:genre :album :track))
      (:artist '(:artist :album :track))
      (:album  '(:album :track))
      (:song   '(:song)))))
```

The function `shuffle-playlist`, called by the Web interface code when the user selects a new shuffle mode, works in a similar fashion except it doesn't need to change the value of `ordering`. Thus, when `shuffle-playlist` is called with a `shuffle` of `:none`, the playlist goes back to being sorted according to the most recent ordering. Shuffling by songs is simple--just call `shuffle-table` on `songs-table`. Shuffling by albums is a bit more involved but still not rocket science.

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))

(defun shuffle-by-song (playlist)
  (shuffle-table (songs-table playlist)))

(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist (column-value album-row :album)))
        (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))

(defun shuffled-album-names (playlist)
  (shuffle-table
   (select
    :columns :album
    :from (songs-table playlist)
    :distinct t)))

(defun songs-for-album (playlist album)
  (select
   :from (songs-table playlist)
   :where (matching (songs-table playlist) :album album)
   :order-by :track))
```

The last manipulation you need to support is setting the playlist's repeat mode. Most of the time you don't need to take any extra action when setting `repeat`--its value comes into play only in

`maybe-move-to-next-song`. However, you need to update the `current-song` as a result of changing `repeat` in one situation, namely, if `current-idx` is at the end of a nonempty playlist and `repeat` is being changed to `:song` or `:all`. In that case, you want to continue playing, either repeating the last song or starting at the beginning of the playlist. So, you should define an `:after` method on the generic function `(setf repeat)`.

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
    (ecase value
      (:song (setf (current-idx playlist) (1- (table-size (songs-table playlist)))))
      (:none)
      (:all  (setf (current-idx playlist) 0)))
    (update-current-if-necessary playlist)))
```

Now you have all the underlying bits you need. All that remains is the code that will provide a Web-based user interface for browsing the MP3 database and manipulating playlists. The interface will consist of three main functions defined with `define-url-function`: one for browsing the song database, one for viewing and manipulating a single playlist, and one for listing all the available playlists.

But before you get to writing these three functions, you need to start with some helper functions and HTML macros that they'll use.

Query Parameter Types

Since you'll be using `define-url-function`, you need to define a few methods on the `string->type` generic function from Chapter 28 that `define-url-function` uses to convert string query parameters into Lisp objects. In this application, you'll need methods to convert strings to integers, keyword symbols, and a list of values.

The first two are quite simple.

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))

(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

The last `string->type` method is slightly more complex. For reasons I'll get to in a moment, you'll need to generate pages that display a form that contains a hidden field whose value is a list of strings. Since you're responsible for generating the value in the hidden field *and* for parsing it when it comes back, you can use whatever encoding is convenient. You could use the functions **`WRITE-TO-STRING`** and **`READ-FROM-STRING`**, which use the Lisp printer and reader to write and read data to and from strings, except the printed representation of strings can contain quotation marks and other characters that may cause problems when embedded in the value attribute of an `INPUT` element. So, you'll need to escape those characters somehow. Rather than trying to come up with your own escaping scheme, you can just use base 64, an encoding commonly used to protect binary data sent through e-mail. AllegroServe comes with two functions, `base64-encode` and `base64-decode`, that do the encoding and decoding for you, so all you have to do is write a pair of functions: one that encodes a Lisp object by

converting it to a readable string with **WRITE-TO-STRING** and then base 64 encoding it and, conversely, another to decode such a string by base 64 decoding it and passing the result to **READ-FROM-STRING**. You'll want to wrap the calls to **WRITE-TO-STRING** and **READ-FROM-STRING** in **WITH-STANDARD-IO-SYNTAX** to make sure all the variables that affect the printer and reader are set to their standard values. However, because you're going to be reading data that's coming in from the network, you'll definitely want to turn off one feature of the reader--the ability to evaluate arbitrary Lisp code while reading![4] You can define your own macro `with-safe-io-syntax`, which wraps its body forms in **WITH-STANDARD-IO-SYNTAX** wrapped around a **LET** that binds ***READ-EVAL*** to **NIL**.

```
(defmacro with-safe-io-syntax (&body body)
  `(with-standard-io-syntax
     (let ((*read-eval* nil))
       ,@body)))
```

Then the encoding and decoding functions are trivial.

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj))))

(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

Finally, you can use these functions to define a method on `string->type` that defines the conversion for the query parameter type `base64-list`.

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

Boilerplate HTML

Next you need to define some HTML macros and helper functions to make it easy to give the different pages in the application a consistent look and feel. You can start with an HTML macro that defines the basic structure of a page in the application.

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
     (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
     (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

You should define `standard-header` and `standard-footer` as separate functions for two reasons. First, during development you can redefine those functions and see the effect immediately without having to recompile functions that use the `:mp3-browser-page` macro. Second, it turns out that one of the pages you'll write later won't be defined with `:mp3-browser-page` but will still need the standard header and footers. They look like this:

```
(defparameter *r* 25)

(defun standard-header ()
```

```
    (html
     ((:p :class "toolbar")
      "[" (:a :href (link "/browse" :what "genre") "All genres") "] "
      "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "] "
      "[" (:a :href (link "/browse" :what "artist") "All artists") "] "
      "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists") "] "
      "[" (:a :href (link "/browse" :what "album") "All albums") "] "
      "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "] "
      "[" (:a :href (link "/browse" :what "song" :random *r*) "Random songs") "] "
      "[" (:a :href (link "/playlist") "Playlist") "] "
      "[" (:a :href (link "/all-playlists") "All playlists") "]")))

(defun standard-footer ()
  (html (:hr) ((:p :class "footer") "MP3 Browser v" *major-version* "." *minor-version*)))
```

A couple of smaller HTML macros and helper functions automate other common patterns. The `:table-row` HTML macro makes it easier to generate the HTML for a single row of a table. It uses a feature of FOO that I'll discuss in Chapter 31, an `&attributes` parameter, which causes uses of the macro to be parsed just like normal s-expression HTML forms, with any attributes gathered into a list that will be bound to the `&attributes` parameter. It looks like this:

```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

And the `link` function generates a URL back into the application to be used as the `HREF` attribute with an `A` element, building a query string out of a set of keyword/value pairs and making sure all special characters are properly escaped. For instance, instead of writing this:

```
(:a :href "browse?what=artist&genre=Rhythm+%26+Blues" "Artists")
```

you can write the following:

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

It looks like this:

```
(defun link (target &rest attributes)
  (html
    (:attribute
     (:format "~a~@[?~{~(~a~)=~a~^&~}~]" target (mapcar #'urlencode attributes)))))
```

To URL encode the keys and values, you use the helper function `urlencode`, which is a wrapper around the function `encode-form-urlencoded`, which is a nonpublic function from AllegroServe. This is--on one hand--bad form; since the name `encode-form-urlencoded` isn't exported from `NET.ASERVE`, it's possible that `encode-form-urlencoded` may go away or get renamed out from under you. On the other hand, using this unexported symbol for the time being lets you get work done for the moment; by wrapping `encode-form-urlencoded` in your own function, you isolate the crufty code to one function, which you could rewrite if you had to.

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

Finally, you need the CSS style sheet `mp3-browser.css` used by `:mp3-browser-page`. Since there's nothing dynamic about it, it's probably easiest to just publish a static file with `publish-file`.

```
(publish-file :path "/mp3-browser.css"  :file filename :content-type "text/css")
```

A sample style sheet is included with the source code for this chapter on the book's Web site. You'll define a function, at the end of this chapter, that starts the MP3 browser application. It'll take care of, among other things, publishing this file.

The Browse Page

The first URL function will generate a page for browsing the MP3 database. Its query parameters will tell it what kind of thing the user is browsing and provide the criteria of what elements of the database they're interested in. It'll give them a way to select database entries that match a specific genre, artist, or album. In the interest of serendipity, you can also provide a way to select a random subset of matching items. When the user is browsing at the level of individual songs, the title of the song will be a link that causes that song to be added to the playlist. Otherwise, each item will be presented with links that let the user browse the listed item by some other category. For example, if the user is browsing genres, the entry "Blues" will contain links to browse all albums, artists, and songs in the genre Blues. Additionally, the browse page will feature an "Add all" button that adds every song matching the page's criteria to the user's playlist. The function looks like this:

```
(define-url-function browse
    (request (what keyword :genre) genre artist album (random integer))

  (let* ((values (values-for-page what genre artist album random))
         (title (browse-page-title what random genre artist album))
         (single-column (if (eql what :song) :file what))
         (values-string (values->base-64 single-column values)))
    (html
     (:mp3-browser-page
      (:title title)
      ((:form :method "POST" :action "playlist")
       (:input :name "values" :type "hidden" :value values-string)
       (:input :name "what" :type "hidden" :value single-column)
       (:input :name "action" :type "hidden" :value :add-songs)
       (:input :name "submit" :type "submit" :value "Add all"))
      (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

This function starts by using the function `values-for-page` to get a table containing the values it needs to present. When the user is browsing by song--when the `what` parameter is `:song`--you want to select complete rows from the database. But when they're browsing by genre, artist, or album, you want to select only the distinct values for the given category. The database function `select` does most of the heavy lifting, with `values-for-page` mostly responsible for passing the right arguments depending on the value of `what`. This is also where you select a random subset of the matching rows if necessary.

```
(defun values-for-page (what genre artist album random)
  (let ((values
         (select
          :from *mp3s*
          :columns (if (eql what :song) t what)
          :where (matching *mp3s* :genre genre :artist artist :album album)
          :distinct (not (eql what :song))
          :order-by (if (eql what :song) '(:album :track) what))))
    (if random (random-selection values random) values)))
```

To generate the title for the browse page, you pass the browsing criteria to the following function, `browse-page-title`:

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre  (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a " artist))
      (when album  (format s " on album ~a" album)))))
```

Once you have the values you want to present, you need to do two things with them. The main task, of course, is to present them, which happens in the `do-rows` loop, leaving the rendering of each row to the function `list-item-for-page`. That function renders `:song` rows one way and all other kinds another way.

```
(defun list-item-for-page (what row)
  (if (eql what :song)
    (with-column-values (song file album artist genre) row
      (html
        (:li
         (:a :href (link "playlist" :file file :action "add-songs") (:b song)) " from "
         (:a :href (link "browse"  :what :song :album  album) album) " by "
         (:a :href (link "browse" :what :song :artist artist) artist) " in genre "
         (:a :href (link "browse"  :what :song :genre  genre) genre))))
    (let ((value (column-value row what)))
      (html
       (:li value " - "
            (browse-link :genre  what value)
            (browse-link :artist what value)
            (browse-link :album  what value)
            (browse-link :song   what value))))))

(defun browse-link (new-what what value)
  (unless (eql new-what what)
    (html
     "["
     (:a :href (link "browse" :what new-what what value) (:format "~(~as~)" new-what))
     "] ")))
```

The other thing on the `browse` page is a form with several hidden `INPUT` fields and an "Add all" submit button. You need to use an HTML form instead of a regular link to keep the application stateless--to make sure all the information needed to respond to a request comes in the request itself. Because the browse page results can be partially random, you need to submit a fair bit of data for the server to be able to reconstitute the list of songs to add to the playlist. If you didn't allow the browse page to return randomly generated results, you wouldn't need much data--you could just submit a request to add songs with whatever search criteria the browse page used. But if you added songs that way, with criteria that included a `random` argument, then you'd end up adding a different set of random songs than the user was looking at on the page when they hit the "Add all" button.

The solution you'll use is to send back a form that has enough information stashed away in a hidden `INPUT` element to allow the server to reconstitute the list of songs matching the browse page criteria. That information is the list of values returned by `values-for-page` and the value of the `what` parameter. This is where you use the `base64-list` parameter type; the function `values->base64` extracts the values of a specified column from the table returned

by `values-for-page` into a list and then makes a base 64-encoded string out of that list to embed in the form.

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

When that parameter comes back as the value of the `values` query parameter to a URL function that declares `values` to be of type `base-64-list`, it'll be automatically converted back to a list. As you'll see in a moment, that list can then be used to construct a query that'll return the correct list of songs.[5] When you're browsing by `:song`, you use the values from the `:file` column since they uniquely identify the actual songs while the song names may not.

The Playlist

This brings me to the next URL function, `playlist`. This is the most complex page of the three--it's responsible for displaying the current contents of the user's playlist as well as for providing the interface to manipulate the playlist. But with most of the tedious bookkeeping handled by `define-url-function`, it's not too hard to see how `playlist` works. Here's the beginning of the definition, with just the parameter list:

```
(define-url-function playlist
    (request
     (playlist-id string (playlist-id request) :package)
     (action keyword)      ; Playlist manipulation action
     (what keyword :file)  ; for :add-songs action
     (values base-64-list) ;             "
     file                  ; for :add-songs and :delete-songs actions
     genre                 ; for :delete-songs action
     artist                ;             "
     album                 ;             "
     (order-by keyword)    ; for :sort action
     (shuffle keyword)     ; for :shuffle action
     (repeat keyword))     ; for :set-repeat action
```

In addition to the obligatory `request` parameter, `playlist` takes a number of query parameters. The most important in some ways is `playlist-id`, which identifies which `playlist` object the page should display and manipulate. For this parameter, you can take advantage of `define-url-function`'s "sticky parameter" feature. Normally, the `playlist-id` won't be supplied explicitly, defaulting to the value returned by the `playlist-id` function, namely, the IP address of the client machine on which the browser is running. However, users can also manipulate their playlists from different machines than the ones running their MP3 clients by allowing this value to be explicitly specified. And if it's specified once, `define-url-function` will arrange for it to "stick" by setting a cookie in the browser. Later you'll define a URL function that generates a list of all existing playlists, which users can use to pick a playlist other than the one for the machines they're browsing from.

The `action` parameter specifies some action to take on the user's playlist object. The value of this parameter, which will be converted to a keyword symbol for you, can be `:add-songs`, `:delete-songs`, `:clear`, `:sort`, `:shuffle`, or `:set-repeat`. The `:add-songs`

action is used by the "Add all" button in the browse page and also by the links used to add individual songs. The other actions are used by the links on the playlist page itself.

The `file`, `what`, and `values` parameters are used with the `:add-songs` action. By declaring `values` to be of type `base-64-list`, the `define-url-function` infrastructure will take care of decoding the value submitted by the "Add all" form. The other parameters are used with other actions as noted in the comments.

Now let's look at the body of `playlist`. The first thing you need to do is use the `playlist-id` to look up the queue object and then acquire the playlist's lock with the following two lines:

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist)
```

Since `lookup-playlist` will create a new playlist if necessary, this will always return a `playlist` object. Then you take care of any necessary queue manipulation, dispatching on the value of the `action` parameter in order to call one of the `playlist` functions.

```
(case action
  (:add-songs      (add-songs playlist what (or values (list file))))
  (:delete-songs   (delete-songs
                    playlist
                    :file file :genre genre
                    :artist artist :album album))
  (:clear          (clear-playlist playlist))
  (:sort           (sort-playlist playlist order-by))
  (:shuffle        (shuffle-playlist playlist shuffle))
  (:set-repeat     (setf (repeat playlist) repeat)))
```

All that's left of the `playlist` function is the actual HTML generation. Again, you can use the `:mp3-browser-page` HTML macro to make sure the basic form of the page matches the other pages in the application, though this time you pass **`NIL`** to the `:header` argument in order to leave out the `H1` header. Here's the rest of the function:

```
(html
 (:mp3-browser-page
  (:title (:format "Playlist - ~a" (id playlist)) :header nil)
  (playlist-toolbar playlist)
  (if (empty-p playlist)
    (html (:p (:i "Empty.")))
    (html
      ((:table :class "playlist")
       (:table-row "#" "Song" "Album" "Artist" "Genre")
       (let ((idx 0)
             (current-idx (current-idx playlist)))
         (do-rows (row (songs-table playlist))
           (with-column-values (track file song album artist genre) row
             (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
               (html
                 ((:table-row :class row-style)
                  track
                  (:progn song   (delete-songs-link :file file))
                  (:progn album  (delete-songs-link :album album))
                  (:progn artist (delete-songs-link :artist artist))
                  (:progn genre  (delete-songs-link :genre genre)))))
             (incf idx)))))))))))
```

The function `playlist-toolbar` generates a toolbar containing links to `playlist` to perform the various `:action` manipulations. And `delete-songs-link` generates a link to

`playlist` with the `:action` parameter set to `:delete-songs` and the appropriate arguments to delete an individual file, or all files on an album, by a particular artist or in a specific genre.

```
(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
     (:p :class "playlist-toolbar"
         (:i "Sort by:")
         " [ "
         (sort-playlist-button "genre" current-sort) " | "
         (sort-playlist-button "artist" current-sort) " | "
         (sort-playlist-button "album" current-sort) " | "
         (sort-playlist-button "song" current-sort) " ] "
         (:i "Shuffle by:")
         " [ "
         (playlist-shuffle-button "none" current-shuffle) " | "
         (playlist-shuffle-button "song" current-shuffle) " | "
         (playlist-shuffle-button "album" current-shuffle) " ] "
         (:i "Repeat:")
         " [ "
         (playlist-repeat-button "none" current-repeat) " | "
         (playlist-repeat-button "song" current-repeat) " | "
         (playlist-repeat-button "all" current-repeat) " ] "
         "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] "))))

(defun playlist-button (action argument new-value current-value)
  (let ((label (string-capitalize new-value)))
    (if (string-equal new-value current-value)
      (html (:b label))
      (html (:a :href (link "playlist" :action action argument new-value) label)))))

(defun sort-playlist-button (order-by current-sort)
  (playlist-button :sort :order-by order-by current-sort))

(defun playlist-shuffle-button (shuffle current-shuffle)
  (playlist-button :shuffle :shuffle shuffle current-shuffle))

(defun playlist-repeat-button (repeat current-repeat)
  (playlist-button :set-repeat :repeat repeat current-repeat))

(defun delete-songs-link (what value)
  (html " [" (:a :href (link "playlist" :action :delete-songs what value) "x") "]"))
```

Finding a Playlist

The last of the three URL functions is the simplest. It presents a table listing all the playlists that have been created. Ordinarily users won't need to use this page, but during development it gives you a useful view into the state of the system. It also provides the mechanism to choose a different playlist--each playlist ID is a link to the `playlist` page with an explicit `playlist-id` query parameter, which will then be made sticky by the `playlist` URL function. Note that you need to acquire the `*playlists-lock*` to make sure the `*playlists*` hash table doesn't change out from under you while you're iterating over it.

```
(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   ((:table :class "all-playlists")
    (:table-row "Playlist" "# Songs" "Most recent user agent")
    (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
           (html
             (:table-row
```

```
                (:a :href (link "playlist" :playlist-id (id playlist)) (:print (id playlist
                (:print (table-size (songs-table playlist)))
                (:print (user-agent playlist)))))))))
```

Running the App

And that's it. To use this app, you just need to load the MP3 database with the `load-database` function from Chapter 27, publish the CSS style sheet, set `*song-source-type*` to `playlist` so `find-song-source` uses playlists instead of the singleton song source defined in the previous chapter, and start AllegroServe. The following function takes care of all these steps for you, after you fill in appropriate values for the two parameters `*mp3-dir*`, which is the root directory of your MP3 collection, and `*mp3-css*`, the filename of the CSS style sheet:

```
(defparameter *mp3-dir* ...)

(defparameter *mp3-css* ...)

(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css"  :file *mp3-css* :content-type "text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001))
```

When you invoke this function, it will print dots while it loads the ID3 information from your ID3 files. Then you can point your MP3 client at this URL:

```
http://localhost:2001/stream.mp3
```

and point your browser at some good starting place, such as this:

```
http://localhost:2001/browse
```

which will let you start browsing by the default category, Genre. After you've added some songs to the playlist, you can press Play on the MP3 client, and it should start playing the first song.

Obviously, you could improve the user interface in any of a number of ways--for instance, if you have a lot of MP3s in your library, it might be useful to be able to browse artists or albums by the first letter of their names. Or maybe you could add a "Play whole album" button to the playlist page that causes the playlist to immediately put all the songs from the same album as the currently playing song at the top of the playlist. Or you could change the `playlist` class, so instead of playing silence when there are no songs queued up, it picks a random song from the database. But all those ideas fall in the realm of application design, which isn't really the topic of this book. Instead, the next two chapters will drop back to the level of software infrastructure to cover how the FOO HTML generation library works.

---

[1]The intricacies of concurrent programming are beyond the scope of this book. The basic idea is that if you have multiple threads of control--as you will in this application with some threads running the `shoutcast` function and other threads responding to requests from the browser--then you need to make sure only one thread at a time manipulates an object in order to prevent one thread from seeing the object in an inconsistent state while another thread is working on it. In this function, for instance, if two new MP3 clients are connecting at the same time, they'd both try to add an entry to `*playlists*` and might interfere with each

other. The `with-process-lock` ensures that each thread gets exclusive access to the hash table for long enough to do the work it needs to do.

[2]This approach also assumes that every client machine has a unique IP address. This assumption should hold as long as all the users are on the same LAN but may not hold if clients are connecting from behind a firewall that does network address translation. Deploying this application outside a LAN will require some modifications, but if you want to deploy this application to the wider Internet, you'd better know enough about networking to figure out an appropriate scheme yourself.

[3]Unfortunately, because of licensing issues around the MP3 format, it's not clear that it's legal for me to provide you with such an MP3 without paying licensing fees to Fraunhofer IIS. I got mine as part of the software that came with my Slimp3 from Slim Devices. You can grab it from their Subversion repository via the Web at `http://svn.slimdevices.com/*checkout*/trunk/server/ HTML/EN/html/silentpacket.mp3?rev=2`. Or buy a Squeezebox, the new, wireless version of Slimp3, and you'll get `silentpacket.mp3` as part of the software that comes with it. Or find an MP3 of John Cage's piece *4′33″*.

[4]The reader supports a bit of syntax, `#.`, that causes the following s-expression to be evaluated at read time. This is occasionally useful in source code but obviously opens a big security hole when you read untrusted data. However, you can turn off this syntax by setting **`*READ-EVAL*`** to **`NIL`**, which will cause the reader to signal an error if it encounters `#.`.

[5]This solution has its drawbacks--if a `browse` page returns a lot of results, a fair bit of data is going back and forth under the covers. Also, the database queries aren't necessarily the most efficient. But it does keep the application stateless. An alternative approach is to squirrel away, on the server side, information about the results returned by `browse` and then, when a request to add songs come in, find the appropriate bit of information in order to re-create the correct set of songs. For instance, you could just save the values list instead of sending it back in the form. Or you could copy the **`RANDOM-STATE`** object before you generate the browse results so you can later re-create the same "random" results. But this approach causes its own problems. For instance, you'd then need to worry about when you can get rid of the squirreled-away information; you never know when the user might hit the Back button on their browser to return to an old browse page and then hit the "Add all" button. Welcome to the wonderful world of Web programming.