5

Writing Form-Based Applications

CHAPTER 4, "GENERATING AND PROCESSING FORMS," covered the general techniques involved in creating forms. In this chapter, we'll use those techniques to write several types of form-based applications, and in Chapter 6, "Automating the Form-Handling Process," we'll discuss how you can use information in your database to help you handle forms automatically.

Web programming includes such a diverse range of applications that we can't hope to cover more than a fraction of the possibilities. Nevertheless, there are several recurring issues, and the applications in this chapter illustrate a number of useful techniques that you should be able to apply to many of your own projects. Therefore, although useful in their own right, these applications are not just ends in themselves—they serve an illustrative purpose as well. The projects we'll tackle, and some of the techniques they involve, are as follows:

- A product-registration script that enables customers to register purchases online by visiting your Web site rather than by mailing in a paper form. It shows how to generate and validate a form on the basis of specifications stored in a data structure.

- A guestbook. I guess every Web programming book has to have a guestbook, so this one does, too. However, our version serves only as a means by which to demonstrate how to incorporate email capabilities into your scripts. The guestbook itself uses email to help keep you apprised of new entries, and we'll cover other ways you can use mail capabilities in your applications. The section also discusses how to set up jobs that execute according to predetermined schedule.
- A giveaway contest application that enables visitors to your site to submit contest entries. This section covers some basic fraud-detection techniques to help combat ballot-box stuffing, selection of random entries to choose contest winners, and entry summary and expiration methods.
- A simple poll. We'll develop a script that uses a form to present candidates users can vote for, and that displays a results page showing the current vote totals. The application uses MySQL to count the votes, and the results page uses the current totals in the database so that the results shown are always up to date.
- Image-storage and image-display scripts. Images are an integral part of many Web applications, which necessitates a method for getting them to your server host and accessing them from within your scripts. To provide support for image use, we'll illustrate how to load images into MySQL two ways: over the Web using a form containing a file-upload field, and from the command line. In addition, this section shows how to display images in Web pages by pulling them from your database.
- An application that enables you to construct an electronic greeting card interactively and notify the recipient that it's waiting. The card is stored as a database record so that it can be retrieved and displayed later for the recipient. The application also notifies you when the recipient views the card. This application incorporates image-display capabilities and shows how to implement multiple-stage record construction, how to trigger a notification when a record's status changes, and how to handle removal of expired cards.

The applications in this chapter have very different purposes, but share certain common characteristics. Generally, you'll find that form-based programs involve the following steps, although the steps vary in complexity from application to application, depending on your goals and requirements:

- Generate a form to solicit the information you want to collect from the user. Some forms are relatively trivial: Our polling application presents a form containing nothing more than a set of radio buttons listing the candidates and a Submit button. The user clicks one time to pick a candidate, a second time to submit the vote, and that's it. Other forms are more extensive: The product-registration application has many fields because we need to gather information about both the product being registered and the user who's registering it.

- Validate the form's contents when the user fills it in and submits it. Form validation can be minimal or quite extensive. You may have fields that are required but were left blank by the user, or fields that must contain a certain kind of information but were filled in incorrectly. In such cases, the user must submit additional or revised information. You'll find it necessary to inform the user that the form cannot be processed, as well as what should be done to correct any problems. We'll demonstrate several feedback techniques over the course of the chapter.
- Store the form submission. Some applications store form information in a file or mail it somewhere for further processing. This being a book on MySQL, we will of course focus on using a database as the primary storage mechanism for each application.
- Generally, you provide some kind of feedback to the user after a form submission has been received and processed. This can be quite simple, such as a thank you message expressing appreciation for the user's participation in a poll or survey, or a brief acknowledgement that the submission was received. Or you may redisplay the information to provide confirmation to the user that it was received properly.

A small reminder before you proceed: You'll find the source code for the scripts developed here in the `webdb` software distribution that accompanies this book. You can get it at the following Web site:

```
http://www.kitebird.com/mysql-perl/
```

You may find it useful to install each script and try it out first before you read the section that describes how it works.

Product Registration

Registration applications serve many purposes. You can use them to allow people to register products they have purchased, sign up for conferences, request a catalog, add themselves to a mailing list, and so forth. The obvious advantage over paper forms for you when you receive a registration from someone is that you needn't re-key the information to get it into your database—the Web application inserts it for you. An advantage for users is instant transmission of their registration information without having to dig up a stamp or put the registration form in the mail.

In this section, we'll write a script, `prod_reg.pl`, that collects product registrations over the Web. Generally, this kind of registration form has, at a minimum, fields to identify the product and the customer, and our application will confine itself to gathering that kind of information. Many registration forms have additional fields for demographic information such as household income, type of employment, or how the product will be used. We'll skip that stuff; you can add it later if you like.

To process registrations, we need a database table in which to store registration records and an application that collects information from customers and inserts records into the table. The table we'll use looks like this:

```
CREATE TABLE prod_reg
(
    # record identification information: unique record ID and time of
    # creation
    id          INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    t           TIMESTAMP,

    # product identification information: serial number, purchase date,
    # where purchased
    serial      VARCHAR(30),
    purch_date  VARCHAR(20),
    store       VARCHAR(30),

    # customer identification information: name (last, first, middle
    # initial), postal address, telephone, email address
    last_name   VARCHAR(30),
    first_name  VARCHAR(30),
    initial     VARCHAR(5),
    street      VARCHAR(30),
    city        VARCHAR(30),
    state       CHAR(2),
    zip         VARCHAR(10),
    telephone   VARCHAR(20),
    email       VARCHAR(60)
)
```

The `prod_reg` table contains columns that serve to identify the record itself as well as the product and the customer. The first two columns, `id` and `t`, provide a unique identification number for each record and indicate when records are created. We can have MySQL supply values for these columns automatically when we create new records; therefore, they need not be represented in the product-registration form.[1] The rest of the columns describe the product (serial number, purchase date, place of purchase) or the customer (all the other columns). The user must supply these product and customer values, so the form will contain corresponding fields for all of them.

1. Using a `TIMESTAMP` enables us to have MySQL fill in the time automatically when we create new records. If you expect to edit records later, however, it wouldn't be an appropriate choice because MySQL would update the value whenever you change a record. In that case, you should pick `DATE` or `DATETIME` and set the `t` column to `CURRENT_DATE` or `NOW()` at record-creation time.

Our application fits into the product-registration process as follows:

- When a customer purchases one of our products, it will include a paper registration form that can be filled in and mailed the traditional way. However, the form also will have a note:

  ```
  If you prefer to register online, visit our Web site at:
  http://www.snake.net/cgi-perl/prod_reg.pl
  ```

- When the customer goes to the Web site (assuming a preference for registering online rather than completing a paper form), the application presents a form for collecting registration information that reflects the structure of our `prod_reg` table. (The form will contain fields corresponding to each of the table's columns except `id` and `t`.) The customer completes the registration by filling in the form, and then selects the Submit button to send the information back to the Web server.
- The application examines the form to make sure all required fields (such as the product serial number) have been provided. If not, we display some feedback to the user indicating that additional information is necessary and redisplay the form so that the customer can complete it properly. Otherwise, we use the form contents to create a new record in the `prod_reg` table and display a confirmation page to the customer indicating that the registration was received.

From this description, we can see that our application needs to generate a form, extract and validate the contents of the submitted form, insert registration information into the database, and display confirmation to the user. (Not coincidentally, these activities correspond to the steps I mentioned earlier in the chapter as those that are common to many form-based applications.)

Designing the Form

For the `prod_reg.pl` application, we'll use a form that consists entirely of plain-text input fields. That's a pretty regular field structure, so instead of writing out a lot of separate `textfield()` calls to generate the fields, let's try a different approach. We can specify form information as an array of field descriptions:

```
my @field_list =
(
    { name => "serial", label => "Serial number:", size => 30, req => 1 },
    { name => "purch_date", label => "Purchase date:", size => 20, req => 1 },
    { name => "store", label => "Where purchased:", size => 30 },
    { name => "last_name", label => "Last name:", size => 30, req => 1 },
    { name => "first_name", label => "First name:", size => 30, req => 1 },
    { name => "initial", label => "Initial:", size => 5 },
    { name => "street", label => "Street:", size => 30 },
    { name => "city", label => "City:", size => 30 },
    { name => "state", label => "State:", size => 2 },
    { name => "zip", label => "Zip code:", size => 10 },
```

continues

continued

```
    { name => "telephone", label => "Telephone:", size => 20 },
    { name => "email", label => "Email address:", size => 60 }
);
```

Each item in the `@field_list` array is a hash that describes a single field. This array serves multiple purposes:

- It helps us generate the entry form. The `name`, `label`, and `size` attributes specify a field name, the label to display next to it (so the user knows what to enter into the field), and a field size. Field names correspond to the names of the underlying table columns, to make it easy to associate form values with the appropriate table columns at record-creation time.
- The array helps us validate submitted form contents when a user sends in a registration: We'll determine whether any given field is required to have a value by checking its `req` attribute.

Because the `@field_list` array applies to both generation and validation of the registration form, we'll pass it as a parameter to multiple functions in the main dispatch logic of our `prod_reg.pl` script:

```
my $choice = lc (param ("choice")); # get choice, lowercased

if ($choice eq "")                  # initial script invocation
{
    display_entry_form (\@field_list);
}
elsif ($choice eq "submit")
{
    process_form (\@field_list);
}
else
{
    print p (escapeHTML ("Logic error, unknown choice: $choice"));
}
```

Generating the Form

The `display_entry_form()` function generates the registration form by iterating through the field list and using the information contained in each list element to construct a call to `textfield()`. Then it adds a Submit button at the end:

```
sub display_entry_form
{
my $field_ref = shift;      # reference to field list
```

```
    print start_form (-action => url ());
    print p ("Please enter your product registration information,\n"
            . "then select the " . strong ("Submit") . " button.");
    foreach my $f (@{$field_ref})
    {
        print escapeHTML ($f->{label}), " ",
            textfield (-name => $f->{name}, -size => $f->{size}),
            br (), "\n";
    }
    print submit (-name => "choice", -value => "Submit"),
            end_form ();
}
```

It isn't necessary to use `escapeHTML()` to encode values passed to `textfield()` because CGI.pm automatically encodes parameters to functions that generate form elements. However, the labels displayed next to the input fields are just static text, so we need to encode them in case they contain any special characters.

One problem with the preceding code is that it produces a visually distracting form. The labels have different lengths, so the input fields that appear to the right of the labels won't line up vertically. We can provide a more regular layout by arranging labels and input fields within columns of an HTML table. Here's a modified version of `display_entry_form()` that produces a table:

```
sub display_entry_form
{
my $field_ref = shift;      # reference to field list
my @row;

    print start_form (-action => url ());
    print p ("Please enter your product registration information,\n"
            . "then select the " . strong ("Submit") . " button.");
    foreach my $f (@{$field_ref})
    {
        push (@row, Tr (
                td (escapeHTML ($f->{label})),
                td (textfield (-name => $f->{name}, -size => $f->{size}))
            ));
    }
    print table (@row),
            submit (-name => "choice", -value => "Submit"),
            end_form ();
}
```

Another improvement would be to give the user a hint about which fields are required to have a value. We can use the `req` attribute for this, but first it's necessary to decide how to indicate “this field is required” to the user. There are different ways to accomplish this:

- Print the labels for the required fields differently than for non-required fields. One common technique is to use red text or bold text. Personally, I don’t like red text very much, because it doesn’t look much different from black text to me. (I have defective color vision.) Printing field labels in bold can be problematic if the user has selected a display font for which bold and plain text don’t look much different. For example, I find that if I set the font to Arial 10-point, bold and plain text don’t look any different in browser windows. Under these circumstances, making the text bold communicates no information.
- Add some kind of image next to missing fields. This is visually distinctive but doesn’t work very well if the user has image loading turned off. Also, it requires more complicated logic in generating the form.
- Use an asterisk next to the label for required fields.

We’ll settle on the simplest technique (putting an asterisk next to the label of each required field). It’s also a good idea to modify the introductory text that precedes the form, to indicate what the asterisks signify:

```
sub display_entry_form
{
my $field_ref = shift;      # reference to field list
my @row;

    print start_form (-action => url ());
    print p ("Please enter your product registration information,\n"
            . "then select the " . strong ("Submit") . " button.");
    print p ("(Fields with an asterisk next to the name are required.)\n");
    foreach my $f (@{$field_ref})
    {
        my $label = $f->{label};
        $label .= "*" if $f->{req};     # add asterisk for required fields
        push (@row, Tr (
                td (escapeHTML ($label)),
                td (textfield (-name => $f->{name}, -size => $f->{size}))
            ));
    }
    print table (@row),
            submit (-name => "choice", -value => "Submit"),
            end_form ();
}
```

Processing Form Submissions

After the user fills in and submits the form, prod_reg.pl handles the submission by calling process_form(). A minimal version of this function might just look through the field list to make sure that the user actually provided values in all the required fields. We can do this by checking the req attribute for each element in our list that describes form fields:

```
sub process_form
{
my $field_ref = shift;      # reference to field list

    foreach my $f (@{$field_ref})
    {
        next unless $f->{req};          # skip test if field is not required
        my $value = param ($f->{name}); # get value
        if (!defined ($value) || $value eq "")
        {
            print p ("Hey, you didn't fill in all the required fields!");
            return;
        }
    }

    # add registration to database and display confirmation page
    insert_record ($field_ref);
}
```

As noted, that's a minimal approach. It doesn't tell the user which fields were missing. It also gets fooled if the user just types spaces in the required fields. We certainly can do a better job. Some of the possible improvements here are to trim extraneous whitespace from the fields before checking them, tell the user which elements are required, and redisplay the form if it's incomplete.

How should we handle the problem of providing feedback to the customer about which fields are missing? Here's one approach:

1. Loop through the fields, checking each of them until we find one that's missing.
2. Announce which field is missing and redisplay the form so that the customer can enter the value and resubmit the form.
3. Repeat until all required fields have been supplied.

Here's another:

1. Check all the fields.
2. Announce which fields are missing and redisplay the form so that the customer can correct all the problems and resubmit the form.

The first method of form validation is simpler to implement, but leads to an application that's extremely tedious to use because the user finds out about only one problem at a time. The second method provides more information. To implement it, we need to save up the error messages as we check the fields. Here is a modified version of `process_form()` that does this:

```
sub process_form
{
my $field_ref = shift;      # reference to field list
my (@errors);

    foreach my $f (@{$field_ref})
    {
        next unless $f->{req};          # skip test if field is not required
        my $value = param ($f->{name}); # get value
        $value = "" unless defined ($value);    # convert undef to empty string
        $value =~ s/^\s+//;             # trim leading/trailing whitespace
        $value =~ s/\s+$//;
        push (@errors, $f->{label}) if $value eq "";    # it's missing!
    }
    if (@errors)
    {
        print p ("Some information is missing."
                . " Please fill in the following fields:");
        s/:$// foreach (@errors);   # strip colons from end of labels
        print ul (li (\@errors));
        display_entry_form ($field_ref);    # redisplay entry form
        return;
    }

    # add registration to database and display confirmation page
    insert_record ($field_ref);
}
```

To validate the form, `process_form()` loops through each field and checks its `req` attribute. If the field is not required, no further test is necessary. Otherwise, we have a look at the field value. First, it's converted to an empty string if the value is `undef`.[2] Then any whitespace at the ends of the value is removed so that we don't think a value is present if it contains only spaces.[3] If the result is empty, a required value is missing and we add a message to the `@errors` array.

2. The purpose of converting undefined values to empty strings is just to suppress warnings that would otherwise be triggered by the presence of the `-w` flag on the script's initial `#!/usr/bin/perl` line. Otherwise, attempting to operate on such values results in messages in the error log.

3. I'm actually a bit ambivalent about pointing out that whitespace removal helps us perform better validation. By doing so, I'm giving away one of my own techniques for dealing with intrusive form-based applications that insist on requiring information that I consider private. I find that just entering a space into a field often will fool an application into thinking that I've given it real information!

If @errors is non-empty after the loop terminates, the form is incomplete. process_form() responds by showing the user which fields must be filled in, and then redisplays the form by calling display_entry_form(). Here is an instance where CGI.pm's sticky form behavior is quite valuable. None of the form element-generating calls in display_entry_form() include an override parameter, so when that function redisplays the form, CGI.pm automatically fills in the fields with whatever values the customer just submitted. That enables us to easily display the form in the same browser window as the error messages. The customer can determine from the error messages at the top of the window which fields still need to be filled in, and can enter the required values without clicking the Back button to return to the entry form. (It's of little value to display error messages if we then require the customer to click Back; as soon as the customer does so, the error messages disappear!)

Before we move on to discuss the insert_record() function that actually adds a new record to the prod_reg table, let's make one more change. One thing process_form() does while checking field values is to trim extraneous whitespace. We'd like to use these tidied-up values when inserting the new registration record, but unfortunately they are just discarded because we used a temporary variable for field checking. If we save the values, we can use them later when adding the record. One easy way to do this is based on the fact that Perl enables us to create new hash attributes on the fly: We can create a new value attribute in each field information hash and stuff the trimmed value into it. The resulting field-checking loop follows. Note that we want to save all values (not just the required ones), so the test of the req attribute must be moved from the beginning of the loop to the end:

```
foreach my $f (@{$field_ref})
{
    my $value = param ($f->{name}); # get field value
    $value = "" unless defined ($value);    # convert undef to empty string
    $value =~ s/^\s+//;             # trim leading/trailing whitespace
    $value =~ s/\s+$//;
    $f->{value} = $value;           # save trimmed value
    push (@errors, $f->{label}) if $value eq "" && $f->{req};
}
```

Storing the Registration Record

After a form has been received that passes the form validation process, insert_record() stores the record and displays a confirmation message to the customer. There are a number of ways to construct an INSERT statement. One source of variation is that MySQL supports different syntaxes for INSERT:

```
INSERT INTO prod_reg SET col1 = val1, col2 = val2, ...
INSERT INTO prod_reg (list of columns) VALUES(list of values)
```

Another source of variation is that DBI supports different ways of specifying data values for a query. We can either insert the values directly into the query string or use placeholders.

To use the first `INSERT` syntax just shown and insert the values into the query string, you could do something like this:

```
$stmt = "";
foreach my $f (@{$field_ref})
{
    next if $f->{value} eq "";      # don't bother with empty fields
    $stmt .= "," if $stmt;          # put commas between assignments
    $stmt .= $f->{name} . " = " . $dbh->quote ($f->{value});
}
$stmt = "INSERT INTO prod_reg SET $stmt";   # complete the statement
$dbh->do ($stmt);                           # and execute it
```

The loop ignores empty fields; we'll just let MySQL supply whatever default values the corresponding columns have. If a field is not empty, we place its value into the string to assign it to the appropriate column—after processing it with `quote()` to perform any quoting and escaping that may be necessary, of course.

To use placeholders instead, we still walk through the fields, but this time we put '`?`' placeholder characters into the query string and save the column values so that we can pass them to the `do()` method:

```
$stmt = "";
@placeholder = ();
foreach my $f (@{$field_ref})
{
    next if $f->{value} eq "";      # don't bother with empty fields
    $stmt .= "," if $stmt;          # put commas between assignments
    $stmt .= $f->{name} . " = ?";   # add column name, placeholder
    push (@placeholder, $f->{value});   # save placeholder value
}
$stmt = "INSERT INTO prod_reg SET $stmt";   # complete the statement
$dbh->do ($stmt, undef, @placeholder);      # and execute it
```

Functionally, there isn't much difference between the two ways of constructing the query, and I doubt whether there is much reason to prefer one over the other in terms of performance, either. Take your pick. (If you expect to be issuing zillions of `INSERT` statements, however, you might want to run a few benchmarks to gather some empirical performance numbers.)

Regardless of how you construct the `INSERT` query, it's a good idea after executing it to present some sort of confirmation to the customer. We'll discuss confirmation pages in more detail later, but for now we'll just present a short message and thank the user:

```
print p ("We have received your product registration. Thank you.");
```

The application doesn't check whether the registration record actually was inserted properly. If we want to warn the customer about problems creating the registration record, we should check the return value from `do()`, and then display an appropriate message:

```
$rows = $dbh->do ($stmt, undef, @placeholder);
if ($rows)
{
    print p ("We have received your product registration. Thank you.");
}
else
{
    print p ("Sorry, we were unable to process your product\n"
            . "registration. Please try again later.");
}
```

Another potential problem is that a serial number might be submitted multiple times. This can occur different ways. The customer might click the Back button and then select Submit again mistakenly. Or a customer might enter the serial number incorrectly, duplicating a number already entered by another customer. What then? As written, the `prod_reg.pl` script does nothing special. It just inserts the record as usual. The theory behind this cavalier behavior is that a customer can't really do much about these problems anyway, so why say anything? On the other hand, you might want to expend a little extra effort to provide yourself with information about possible problems. For example, you could run a query periodically to find duplicate serial numbers so that you can examine the appropriate records. Here's a query that finds serial numbers that appear multiple times in the `prod_reg` table:

```
SELECT serial, COUNT(serial) AS count FROM prod_reg
GROUP BY serial HAVING count > 1
```

Enter a registration record, and then click your browser's Back button and the form's Submit button a few times to generate some duplicate records. Then run the preceding query from the `mysql` program to see what kind of output it produces. If you adopt the approach of looking for duplicate values using this query, you'll find that it becomes much slower as your table accumulates more records. That's because the table doesn't have any indexes on it. Use `ALTER TABLE` to add an index on the serial column and the query will run much more quickly:

```
ALTER TABLE prod_reg ADD INDEX (serial)
```

Checking for Empty Value Lists

The `insert_record()` function constructs an `INSERT` statement, but doesn't bother to check whether the list of column assignments following the `SET` keyword is empty. We know it won't be, because our form has several required fields and `insert_record()` isn't called unless they've been filled in. If for some reason you have a form for which no fields are required, however, you should check for an empty column assignment list. Why? Because if the form is empty and you don't check, you'll find yourself constructing and trying to execute an `INSERT` statement that looks like this, which is syntactically invalid:

```
INSERT INTO tbl_name SET
```

If you're determined to insert a new record even when there are no values, you might want to consider using an `INSERT` syntax, which does allow an empty value list:

```
INSERT INTO tbl_name () VALUES()
```

This "empty" `INSERT` syntax is understood by MySQL 3.23.3 and later versions.

Suggested Modifications

`process_form()` displays all the error messages together at the top of the browser window if there are problems with a form submission. Modify the application to display each message immediately below the field to which it applies, to put the messages in closer proximity to the offending fields. You can do this by creating an `error` attribute in `@field_list` elements, and changing `display_entry_form()` to check for and display that attribute. If you want to display a warning icon as well, what would you need to do?

If you have independent information about your products, you may be able to use it to improve the integrity of product registrations. Suppose you have a product-listing table that contains a record of each item that you've manufactured, including its serial number. You could have the `prod_reg.pl` application present an initial "gatekeeper" page that displays a form containing only an input field for the product serial number. When the customer supplies the number, check your product listing to verify that the number is valid, and proceed to the main entry form if so. Otherwise, you know the serial number was entered incorrectly and you can ask the user to double-check the number on the product.

`prod_reg.pl` isn't very rigorous in its validation, other than to make sure required fields aren't missing. Modify it to perform field-specific validation. Make sure the state abbreviation is legal, for example, or verify that if the email field is filled in that the value looks like a valid address. For extra fun, check the purchase date to make sure it looks like a date. (Because dates can be specified many ways, however, you probably should display a note indicating the format you expect date values to follow.)

Using text boxes for all the input fields makes it easy to describe the form using a data structure so that we can generate it and validate it automatically. But some of the fields might be easier for the user to fill in if we had used other field types. For example, the field for entering the state could be represented by a scrolling list containing the possible state abbreviations. (This would also help make sure that users don't enter an incorrect state value.) How would you modify the `@field_list` array and the form-generation code to handle this? Is it worth it?

The application pretty much assumes the postal address style used in the United States. (For example, we have a field for Zip Code, which is hardly a worldwide convention.) How would you generalize the form to be less U.S.-centric?

Modify `insert_record()` to present a more informative confirmation page that shows the customer what information was stored. (You might be surprised how often people recognize that they've entered one or more values incorrectly if you just reformat the information and show it to them again.) If you make this change, you should also provide information about whom the customer should contact to report any errors. An alternative to storing the record in the database before displaying it in a confirmation page is to present the information along with an "Are you sure?" button. If the customer is sure, store the record in the database; otherwise, allow the customer to edit the information further and resubmit it.

Using Email from Within Applications

When I began to outline which applications to discuss in this book, I considered whether to include a guestbook that visitors to your site can "sign" by filling in a form. A guestbook is simple, so it makes a good introductory form-based application that you can write without a lot of work. Also, you can store the comments in a database, which gives it some relevance to MySQL. But guestbooks have been covered in so many books on Web programming that they've become standard fodder, and my initial gut reaction to writing about them was "ugh."

What to do? I went back and forth from one horn of this dilemma to the other several times. Eventually, I decided to include a guestbook application (right here, in fact), but primarily as an excuse to cover something else: how to add email capabilities to scripts. The guestbook is so simple that we don't have to pay much attention to it, yet it provides a meaningful context for discussing email-generation techniques. Dilemma resolved.

Here is the scenario: You want to provide a guestbook so that people who visit your site can comment on it, and you'll allow people to view comments from previous visitors. But you're concerned about people submitting comments that are unsuitable. If someone writes something that contains foul or libelous language, for instance, you'll want to remove it. That means the guestbook needs to be monitored—but you don't want to be bothered having to check it all the time for new entries, or it becomes a burden. One solution to this problem is to have the guestbook itself tell you when it gets a new entry, by sending you an email message containing the content of the entry. That way you receive updates as new entries arrive, just by reading your mail as you normally do. You don't have to keep remembering to visit the guestbook to check for recent additions.

To hold guestbook comments, we'll use the following table. Visitors will provide the `name`, `email`, and `note` columns; we'll provide the `id` and `date` values:

```
CREATE TABLE guest
(
    id      INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    date    DATE NOT NULL,
    name    VARCHAR(50),
    email   VARCHAR(60),
    note    VARCHAR(255)
)
```

The `guest.pl` script that handles the guestbook has three functions: display an entry form, process new entries, and show previous entries. Accordingly, the main program looks like this:

```
print header (),
    start_html (-title => "Sign or View My Guestbook", -bgcolor => "white"),
    h2("Sign or View My Guestbook");

my $choice = lc (param ("choice")); # get choice, lowercased

if ($choice eq "")                  # initial script invocation
```

continues

continued

```
{
    display_entry_form ();
}
elsif ($choice eq "submit")
{
    process_form ();
}
elsif ($choice eq "view comments")
{
    display_comments ();
}
else
{
    print p (escapeHTML ("Logic error, unknown choice: $choice"));
}

print end_html ();
```

The `display_entry_form()` function presents a simple form for entering comments and a button that allows previous comments to be viewed:

```
sub display_entry_form
{
    print start_form (-action => url ()),
        table (
            Tr (
                td ("Name:"),
                td (textfield (-name => "name", -size => 60))
            ),
            Tr (
                td ("Email address:"),
                td (textfield (-name => "name", -size => 60))
            ),
            Tr (
                td ("Note:"),
                td (textarea (-name => "name", -cols => 60, -rows => 4))
            ),
        ),
        br (),
        submit (-name => "choice", -value => "Submit"),
        " ",
        submit (-name => "choice", -value => "View Comments"),
        end_form ();
}
```

If the user selects the View Comments button, the script calls `display_comments()` to look up all the previous comments and display them:

```
sub display_comments
{
my $dbh;
my $sth;
```

```
    $dbh = WebDB::connect ();
    $sth = $dbh->prepare ("SELECT * FROM guest ORDER BY id");
    $sth->execute ();
    while (my $ref = $sth->fetchrow_hashref ())
    {
        print p (escapeHTML ("ID: $ref->{id} Date: $ref->{date}")),
                p (escapeHTML ("Name: $ref->{name}")),
                p (escapeHTML ("Email: $ref->{email}")),
                p (escapeHTML ("Note: $ref->{note}")),
                hr ();
    }
    $sth->finish ();
    $dbh->disconnect ();
}
```

If the visitor fills in the form and selects the Submit button, we extract the form contents, add a new record to the database, and send the guestbook administrator an email message:

```
sub process_form
{
my $dbh;
my $sth;
my $ref;

    # Add new entry to database, then reselect it so we can get the
    # id and date values to use for the mail message.
    $dbh = WebDB::connect ();
    $dbh->do ("INSERT INTO guest (date,name,email,note) VALUES (NOW(),?,?,?)",
                undef,
                param ("name"), param ("email"), param ("note"));
    $sth = $dbh->prepare ("SELECT * FROM guest WHERE id = LAST_INSERT_ID()");
    $sth->execute ();
    $ref = $sth->fetchrow_hashref ();
    $sth->finish ();
    $dbh->disconnect ();

    # display confirmation to user
    print p ("Thank you.");

    # send the entry to the administrator via email
    mail_entry ($ref);
}
```

In the interest of making the application as simple as possible, `process_form()` performs no validation whatsoever. It just adds whatever information the user provided, even if there isn't any. Then, after adding the new entry, the application needs to mail a copy of it to you. However, this information must include the ID number of the entry (you'll need that value so you can specify which record to remove if it's found to be objectionable), and we might as well present the date, too. To look up the record just inserted, we identify it using `LAST_INSERT_ID()`. That function returns the most recent `AUTO_INCREMENT` value created during the current session with the MySQL server, which just happens to be the `id` value of the record just inserted. After we have the record, we're ready to send some email by calling `mail_entry()`. Before describing that function, I'll discuss the general technique for sending mail from a script, because we'll be doing that several places in this book. Then we'll adapt that technique for `guest.pl` and use it within the context of mailing guestbook entries.

Sending Email from Scripts

To send mail from within a Perl script, it's pretty common on UNIX systems to open a pipe to the `sendmail` program and write the message into it like so:

```
open (OUT, "| /usr/lib/sendmail -t")
    or die "Cannot open pipe to sendmail: $!\n";
print OUT "From: $sender\n";
print OUT "To: $recipient\n";
print OUT "Subject: I'm sending you mail\n";
print OUT "\n";     # blank line between headers and message body
print OUT "This is the message body.\n";
close (OUT);    # close pipe to send message
```

There are other mail-sending alternatives. The Perl CPAN provides several modules for this purpose; the one we'll use is `Mail::Sendmail`. One of its advantages is that it doesn't require `sendmail` and therefore works under both UNIX and Windows.[4]

Here is a short script, `testmail.pl`, that illustrates how to use `Mail::Sendmail`:

```
#! /usr/bin/perl -w
# testmail.pl - Send mail using the Mail::Sendmail module

use strict;
use Mail::Sendmail;

my $recipient = "black-hole\@localhost";    # CHANGE THIS!
my $sender = "black-hole\@localhost";       # CHANGE THIS!

# Set up hash containing mail message information
my %mail = (
    From    => $sender,
    To      => $recipient,
    Subject => "I'm sending you mail",
    Message => "This is the message body.\n"
);
sendmail (%mail) or die "sendmail failure sending to $mail{To}: $!\n";

exit (0);
```

4. If `Mail::Sendmail` isn't available on your system, you should obtain and install it before proceeding. (See Appendix A, "Obtaining Software.") You can use one of the other mail modules in the CPAN if you like, but you'll have to adapt the scripts in this book that assume the use of `Mail::Sendmail`.

The script accesses Mail::Sendmail by including the appropriate use statement. Then it sets up a hash containing keys for various parts of the message and passes the hash to the sendmail() function. From, To, Subject, and Message are the most useful attributes, but sendmail() understands others, too. (You should change the $recipient and $sender addresses before you try out this script on your system. However, note that the addresses should include a host name, or sendmail() may reject the message.) For further information, you can read the documentation using this command:

```
% perldoc Mail::Sendmail
```

Based on the preceding discussion, we can write a mail_entry() function for guest.pl that takes care of sending new entries to the administrator. The argument passed to it is the new entry record from the guest table, supplied as a hash reference. mail_entry() uses the contents of the entry to format a simple message body and mails it out. (You'll want to change the To and From addresses before trying out the script on your own site.)

```
sub mail_entry
{
my $ref = shift;
my %mail =
(
    From => "black-hole\@localhost",            # CHANGE THIS!
    To => "black-hole\@localhost",              # CHANGE THIS!
    Subject => "Exciting New Guestbook Entry",
    Message => "
id:     $ref->{id}
date:   $ref->{date}
name:   $ref->{name}
email:  $ref->{email}
note:   $ref->{note}
"
);

    sendmail (%mail);
}
```

Removing Guestbook Entries

Eventually, guest.pl may mail an entry to you that you deem unworthy of inclusion in your guestbook. How do you remove it? One way to delete the entry is to use the mysql program and issue the appropriate query manually:

```
mysql> DELETE FROM guest WHERE id = 37;
```

But it would be more convenient to have a command that requires you to provide only the ID for the entry (or entries) you want to delete:

```
% ./guest_clobber.pl 37 81 92
```

Here's a script you can use for that. There's not much to it, because all it needs to do is construct and run one or more `DELETE` statements:

```
#! /usr/bin/perl -w
# guest_clobber.pl - remove guestbook entries; name IDs on the command line

use strict;
use lib qw(/usr/local/apache/lib/perl);
use WebDB;

die "No arguments given\n" unless @ARGV;
my $dbh = WebDB::connect ();
$dbh->do ("DELETE FROM guest WHERE id = ?", undef, shift (@ARGV)) while @ARGV;
$dbh->disconnect ();

exit (0);
```

Dealing with High Guestbook Traffic

If your guestbook turns out to be quite popular, you may find that it's sending you more mail messages than you care to deal with. In that case, you can employ a different approach: Generate a single "digest" message once per day containing all the entries from the previous day. That means there will be more of a delay before you receive notification about new entries, but you'll get one larger message rather than a bunch of smaller ones. If you want to do this, modify `guest.pl` by ripping out the code that generates email, and then set up a program that runs on schedule each day to identify the relevant entries and mail them to you.

Job scheduling under UNIX involves adding an entry to your `crontab` file that specifies a program you want run by the `cron` program.[5] Details vary slightly among systems; use the following commands to get the particulars for your system about the `cron` program and about `crontab` file editing:

```
% man cron
% man crontab
```

On some systems, a separate command describes the `crontab` file format:

```
% man 5 crontab
```

5. Under Windows, you can use one of the `cron` clones that are available; there are freeware, shareware, and commercial versions. (Appendix A lists where you can obtain these products.)

To schedule a job, add a line to your `crontab` file that specifies when the job should run. If I have a script `guest_digest.pl` installed in my `bin` directory, I can specify that I want it to run each day at 1:00 a.m. with a `crontab` line like this:

```
0 1 * * * /u/paul/bin/guest_digest.pl
```

The first four fields indicate the minute, hour, day, and month the job should run. A '`*`' character indicates execution every applicable interval (for example, the '`*`' characters in the third and fourth fields mean "every day of every month"). The fifth field is used for day-of-week scheduling, with values of `0` through `6` meaning Sunday through Saturday.

Now that we have set up scheduled execution of the `guest_digest.pl` script, I suppose we'd better write it:

```
#! /usr/bin/perl -w
# guest_digest.pl - produce digest of yesterday's guestbook entries

use strict;
use lib qw(/usr/local/apache/lib/perl);
use WebDB;

my $prelude = "Yesterday's guestbook entries:";

my $dbh = WebDB::connect ();
my $sth = $dbh->prepare (
                "SELECT * FROM guest
                WHERE date = DATE_SUB(CURRENT_DATE,INTERVAL 1 DAY)
                ORDER BY id");
$sth->execute ();
while (my $ref = $sth->fetchrow_hashref ())
{
    # print introductory text, but only before the first
    # record, and only if there are any records
    if (defined ($prelude))
    {
        print "$prelude\n";
        undef $prelude;
    }
    print <<EOF;
---------------------------------------------
id:     $ref->{id}
date:   $ref->{date}
name:   $ref->{name}
email:  $ref->{email}
note:   $ref->{note}
EOF
}
$sth->finish ();
$dbh->disconnect ();

exit (0);
```

In the crontab entry, I indicated that guest_digest.pl is in the /u/paul/bin directory, so that's where I'd install the script. For testing, we can run guest_digest.pl manually at the command line:

```
% /u/paul/bin/guest_digest.pl
Yesterday's guestbook entries:
----------------------------------------
id:     57
date:   2001-01-24
name:   Joe Visitor
email:  joev@joevis.net
note:   Here's my pithy comment!
...
```

The script figures out yesterday's date using MySQL's DATE_SUB() function and displays all the entries created on that date. You'll notice that guest_digest.pl doesn't reference Mail::Sendmail and it doesn't call sendmail(). So, you may be wondering how it sends you a daily mail message when it runs under cron. The answer is that guest_digest.pl itself doesn't, but cron does. When cron runs a program specified in a crontab file, it mails any output produced by the program to the crontab owner. Therefore, we're still generating email—we're just not doing so explicitly from the script. If your cron doesn't behave that way and just throws away any job output, you should of course modify guest_digest.pl to mail its output explicitly. In fact, you might decide to do that anyway, because the Subject: header for cron-generated mail isn't especially informative—as you'll see when it sends you a message for the first time.

Suggested Modifications

The guestbook application could stand some improvement. The most obvious shortcoming is that process_form() doesn't perform even the most rudimentary form validation, such as making sure the visitor provided at least a name. Add some code to do that.

As written, when you submit a comment, you cannot view comments by other visitors except by returning from the confirmation page to the entry-form page. Improve the application's navigation options by putting a View Comments button in the confirmation page so that visitors can go right to the past comments page from there.

The page generated by display_comments() will become quite large after your guestbook receives a number of entries. Rewrite that function to either limit the display to the *n* most recent entries, or to present a multiple-page display with each page limited to *n* entries.

Other Uses for Email

Email can be used in many different ways from your scripts. For example, you can email a message to people who fill in a form. This can be useful if you want to provide additional confirmation to users from whom you've received a form submission, in a way that's more permanent than displaying a message in a browser window. (Any information in the window disappears as soon as the user closes it or visits another page.) Suppose a customer uses the product-registration application developed earlier in the chapter and happens to fill in the email field in the form. You could use the address to mail a copy of the registration record to that customer. To provide this capability, add this line to the end of the `insert_record()` function in the `prod_reg.pl` script:

```
mail_confirmation ($field_ref);
```

Then add the following function to the script; it checks whether the email address is present and sends a message to that address if so:

```
sub mail_confirmation
{
my $field_ref = shift;      # reference to field list
my %mail =
(
    From => "black-hole\@localhost",        # CHANGE THIS!
    Subject => "Your product registration information"
);

    # Determine whether or not the email field was filled in
    foreach my $f (@{$field_ref})
    {
        if ($f->{name} eq "email")
        {
            return unless $f->{value};  # no address; do nothing
            $mail{To} = $f->{value};
            last;
        }
    }
    $mail{Message} = "Thank you for registering your product.\n";
    $mail{Message} .= "This is the information we received:\n\n";
    foreach my $f (@{$field_ref})
    {
        $mail{Message} .= "$f->{label} $f->{value}\n";
    }
    sendmail (%mail);
}
```

Note that the `mail_confirmation()` function verifies that the email field is filled in before attempting to send mail, but it doesn't check whether the value actually looks like a valid address. We'll see how to perform that kind of test when we write our next application.

Another way to use email within the application would be to have it send you notification if it notices that a user submits a registration with a serial number that duplicates a number already in the table. This probably indicates a typo on some user's part.

Some uses for email apply not at the time you collect information from your visitors, but later. It's fairly common to put a field in a form allowing a user to sign up to receive announcements. You'd store the address when the form is submitted, and then retrieve all addresses later each time you have an announcement to send.

Running a Giveaway Contest

Some Web sites are fairly stable, with content that doesn't change very much. Suppose, however, that you operate a site that provides frequently changing information, such as a news site for a radio or television station. You probably want people to visit often to see new material, and you may be looking for some sort of attraction to lure them to the site on a recurring basis. Then it hits you: Offer a periodic giveaway contest, where people can enter their name once for each giveaway. This serves multiple purposes:

- It encourages people to visit your site regularly.
- You get people to divulge information that you can use later to send them annoying junk mail telling them how wonderful your site is and urging them to visit even *more* frequently!

Okay, I'm just kidding about that second purpose. I don't condone gathering information and then misusing it, although that is often the reason giveaways are held. (Ever notice how you end up on a new mailing list after you put your business card into one of those "Win a Free Dinner!" fishbowls at a restaurant? Now you know why; those things are really a trap for the unwary.)

There are several aspects to running a giveaway contest such as this:

- Obviously, you have to have prizes to give away. I leave this part up to you.
- You need to collect entries. This can be done using a form-based script.
- You have to perform periodic drawings to determine who the winners are. Initially, we'll write our application to handle a daily giveaway, because the queries are simpler to write. Later we'll consider the issues involved in modifying the application for longer periods of time, such as weekly or monthly.
- You must notify the winners of their good fortune.
- You may want to run summary queries to provide you with information about the popularity of your contest, or perhaps delete old entries.
- When you're ready to retire the application, remember to remove it from the Web site. Otherwise, you'll have one of those hideous "Most recent winner: Jane Doe, February 13, *XXXX*" situations, where *XXXX* is about 10 years ago. A good way to disgust your visitors is to awaken their interest in a contest and then have them realize that it's an ancient promotion that you're no longer maintaining.

We'll discuss how to implement each of these phases of the contest shortly (except for acquiring the prizes and deleting the application), but first we should consider something else: fraud.

Detecting Fraud

One issue to be concerned about when you develop this kind of application is the potential for ballot-box stuffing. You don't want people to submit multiple entries to gain an unfair advantage over other contestants. Detection of fraudulent entries is a very difficult problem to solve in general, because in a Web environment it's difficult to uniquely identify visitors to your site. Fortunately, one thing you have working for you in a giveaway situation is that contestants have an incentive to submit correct information. (Only a silly or stupid contestant would submit an entry containing fake information, because you'd have no way to contact the contestant if you drew that entry as a winner.) You can use this fact to help you determine whether one entry duplicates another.

One general "bad guy" strategy for submitting multiple entries is to enter information each time that is correct yet slightly different from earlier entries. Conversely, a general strategy for defeating these attempts is to remove variation by converting values to a single standard format before storing them. This gives you a better chance of detecting duplicates. The following list discusses some of these techniques:

- Trim leading and trailing whitespace to prevent people from attempting to submit multiple entries using values that differ only in the amount of whitespace:[6]

  ```
  $val =~ s/^\s+//;
  $val =~ s/\s+$//;
  ```

- Convert sequences of multiple whitespace characters to single spaces. This defeats attempts to make values unique that are based on putting differing numbers of spaces between words:

  ```
  $val =~ s/\s+/ /g;
  ```

- Don't be fooled by letter-case variations. "Paul", "PAUL", and "paul" are all the same name, and you'd want to recognize them as such. To that end, don't store text values in case-sensitive table columns, or else convert values to a known case before storing them. For example, you can convert strings to uppercase or lowercase like this:

  ```
  $val = uc ($val);
  $val = lc ($val);
  ```

  `CHAR` and `VARCHAR` columns are not case sensitive. `CHAR BINARY` and `VARCHAR BINARY` are case sensitive, as are `TEXT` and `BLOB` columns. If you do use a column type that is case sensitive, you should force values to a given case before storing them.

6. We used this whitespace-removal technique earlier, but for another reason: to prevent our applications from thinking that a field containing only tabs or spaces has a real value in it.

- Remove punctuation if you can. The following pattern removes all characters but letters and spaces, which converts differing values, such as "John A. Smith" and "John A Smith", to the same value:

  ```
  $val =~ s/[^ a-zA-Z]//g;
  ```

- Convert values that are partially numeric to strictly numeric form when possible. A user might attempt to submit three entries by using variant forms of the same telephone number, such as "123-4567", "123 4567", and "1234567". A string comparison would consider these all different, even though you'd easily recognize them as the same. By stripping out the non-digit characters, all three forms become the same value "1234567". You can do this by performing the following substitution:

  ```
  $val =~ s/\D//g;
  ```

  The same kind of conversion can be useful with other types of values, such as credit card or social security numbers.

We'll use several of these techniques in our giveaway application. They aren't a complete solution to the duplicate-entry problem, but they do help make entries more uniform. These value-transformation operations are likely to be useful in other scripts as well, so we'll add them as utility functions to our `WebDB.pm` module file (see Chapter 2, "Getting Connected—Putting Your Database on the Web"). For example, a function to trim whitespace from the ends of a value can be written like this:

```
sub trim
{
my $str = shift;

    return "" if !defined $str;
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;
    return ($str);
}
```

`trim()` is likely to be the first transformation we apply to field values, so we also can have it convert `undef` to the empty string. That way it takes care of two value-checking operations automatically. To use `trim()`, invoke it as follows:

```
$val = WebDB::trim ($val);
```

The other functions are written in similar fashion; we'll call them `collapse_whitespace()`, `strip_punctuation()`, and `strip_non_digits()`. You'll find all these functions in the version of `WebDB.pm` provided in this book's `webdb` source distribution.[7]

7. If you decide to add other validation functions to `WebDB.pm` and you're using `mod_perl`, remember that you'll need to restart Apache. `mod_perl` notices only changes to scripts that are invoked directly, not changes to library modules.

Making entries more uniform helps us in two ways. First, we can attempt to eliminate as many duplicate entries as possible at the time they are submitted. To do this, we'll require contestants to provide some piece of information that makes entries unique. If you consider telephone numbers unique and want to allow only one entry per person per day, for example, you can construct a unique index on the contestant record table that consists of the entry date and the telephone number. Then if someone attempts to enter twice on the same day, we can toss the second entry by noticing that an entry with the same phone number has already been received. (There are at least three problems with this strategy: A person might have multiple phone numbers and could submit one entry per number; a person might enter once with the area code specified and once without; and people who legitimately share the same phone number, such as household members, are prevented from each submitting separate entries. However, I'm going to ignore these problems and just observe that they illustrate the difficulty of arriving at bulletproof uniqueness criteria. You will of course want to determine specific fraud-detection guidelines for your own situation.)

Second, entries that are more uniform help us each time we draw a winning entry. When we select a specific contestant record, we can use it to locate other entries that are near matches and examine them to see whether they look suspiciously similar to the winning entry.

Designing the Contestant Table

The table for storing contestant entries is not complicated. As usual, we'll have an `AUTO_INCREMENT` column named `id` to which MySQL will assign unique identifier numbers, and a column to record the record-creation time. We'll also have a few columns that identify the contestant:

```
CREATE TABLE contestant
(
    id          INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    entry_date  DATE NOT NULL,
    name        VARCHAR(50) NOT NULL,
    email       VARCHAR(60) NOT NULL,
    telephone   VARCHAR(20) NOT NULL,
    UNIQUE (entry_date, telephone)
)
```

The `contestant` table includes a unique index composed of values in the `entry_date` and `telephone` columns. This enforces the constraint that only one entry per day can be submitted for any given phone number. The date can be assigned by our application, `giveaway.pl`, when it receives an entry. The telephone number must be provided by the contestant, but might be entered using varying formats, so we'll perform some preprocessing on it to increase our chances of duplicate detection.

Collecting Contest Entries

The main logic of the `giveaway.pl` script is pretty much identical to that of `prod_reg.pl`, so it need not be shown here. Basically, it checks the `choice` parameter to see whether it's `empty` or `submit`, and then displays the entry form or processes a submitted form accordingly.

The contestant entry form contains only three text-input fields (contestant's name, email address, and telephone number) and a Submit button. Here again, as with the product-registration application, we can use a hash to describe each field, and then pass the hash array to the form-generating and record-entry functions. The field array looks like this:

```
my @field_list =
(
    { name => "name", label => "Name:" },
    { name => "email", label => "Email address:" },
    { name => "telephone", label => "Telephone:" }
);
```

The array contains field hashes that differ slightly from those we used for the product-registration form. First, there is no `req` hash attribute to indicate which fields must be non-empty. We're going to require all fields to be filled in, so the form-validation code can enforce that requirement uniformly for all fields. Second, we'll make all fields the same length, so there's no need for a `size` attribute, either. The code to generate the entry form ends up much like that for product registrations, with some small modifications:

```
sub display_entry_form
{
my $field_ref = shift;      # reference to field list
my @row;

    print start_form (-action => url ()),
            p ("Please complete the following form to submit your\n"
                . "entry for our giveaway contest. Please, only\n"
                . "one entry per person per day.");
    foreach my $f (@{$field_ref})
    {
        push (@row, Tr (
                td (escapeHTML ($f->{label})),
                td (textfield (-name => $f->{name}, -size => 60 ))
            ));
    }
    print table (@row),
            submit (-name => "choice", -value => "Submit"),
            end_form ();
}
```

When a contestant submits an entry, we'll perform some tests on it. If the entry is unsuitable, we'll redisplay the form with appropriate feedback indicating how to fix the problems. Otherwise, we'll use the form contents to create a new contestant table record and thank the user for participating:

```
sub process_form
{
my $field_ref = shift;      # reference to field list
my @errors;
my ($name, $email, $telephone);
my $dbh;

    # First, make sure all fields have a value
    foreach my $f (@{$field_ref})
    {
        my $val = WebDB::trim (param ($f->{name})); # get trimmed field value
        push (@errors, $f->{label}) if $val eq "";  # it's empty!
        # put modified value back into environment
        param (-name => $f->{name}, -value => $val);
    }
    if (@errors)
    {
        print p ("Some information is missing."
                . " Please fill in the following fields:");
        s/:$// foreach (@errors);   # strip colons from end of labels
        print ul (li (\@errors));           # print column names
        display_entry_form ($field_ref);    # redisplay entry form
        return;
    }

    # Re-extract the modified values from the environment and perform some
    # field-specific checks.  Also, transform values to more uniform format
    # to make it easier to catch duplicates.

    $name = param ("name");
    $email = param ("email");
    $telephone = param ("telephone");

    # Collapse runs of white space, eliminate punctuation in the name field
    $name = WebDB::collapse_whitespace ($name);
    $name = WebDB::strip_punctuation ($name);

    # Check the email value using a rudimentary pattern that requires
    # a user name, an @ character, and a hostname containing at least
    # two components.

    if (!WebDB::looks_like_email ($email))
    {
        push (@errors, "Email address must be in user\@host.name format");
    }
```

```
        # Strip non-digit characters from telephone number
        $telephone = WebDB::strip_non_digits ($telephone);
        if (length ($telephone) < 7)
        {
            push (@errors, "Telephone number must contain at least seven digits");
        }

        if (@errors)
        {
            print p ("Your entry could not be processed. Please\n"
                    . "correct the following problem(s) and resubmit the entry:");
            print ul (li (\@errors));           # print error messages
            display_entry_form ($field_ref);    # redisplay entry form
            return;
        }

        # Everything looks okay.  Insert record, thank user.
        # Use INSERT IGNORE to ignore attempts to enter twice.
        # (If an entry contains entry_date and telephone values
        # that duplicate an existing record, it will be rejected.)

        $dbh = WebDB::connect();
        $dbh->do ("INSERT IGNORE INTO contestant (entry_date,name,email,telephone)
                    VALUES(CURRENT_DATE,?,?,?)",
                undef,
                $name, $email, $telephone);
        $dbh->disconnect ();

        print p ("We have received your entry. Thank you.");
        print p ("We will notify you if you are a winner!");
    }
```

The test to verify that each field has been filled in is similar to what we've done before: Trim whitespace from the ends and check whether the result is empty. To avoid having to strip whitespace again later when performing more specific tests, the loop stores the modified value back into the environment:

```
# put modified value back into environment
param (-name => $f->{name}, -value => $val);
```

That way, when we extract the values again later, we get the already-stripped versions. In effect, we're treating the parameter space as a global clipboard that can be used to pass data around between different phases of an application.[8]

8. We could also have used the technique of storing the stripped variable values in a `value` attribute of the field information hashes, the way we did for the product-registration application. I'm using `param()` here to illustrate another technique that accomplishes the same end without modifying the field hashes.

For fields found to be missing, we collect error messages and display them if the form is not complete. If values were supplied for all fields, further processing is performed on each field according to the type of information it's expected to contain. The field-specific operations are designed to verify that the content matches some expected format and to put values into a standardized format that will help us identify duplicate entries.

For the name, we collapse runs of whitespace characters to single spaces and eliminate punctuation. This isn't necessary for validation of the entry, but it's useful for fraud detection later: After we pick a winning entry, standardizing the `name` values will help us find possible duplicates on the `name` field.

For the `email` value, we call `looks_like_email()`, a utility routine that checks whether a value looks like an email address in `user@host.name` form. Email address validation can be a really complicated task to perform in any exhaustive sense. For example, Friedl (*Mastering Regular Expressions*, by Jeffrey E. F. Friedl) shows a regular expression that works for the general case, but it's 6598 characters long! I prefer something a bit less complex that just weeds out obvious clunkers. We'll require a string containing a username, an '`@`' character, and a domain name consisting of at least two components separated by periods:

```
sub looks_like_email
{
my $str = shift;

    return ($str =~ /^[^@]+@[^.]+\.[^.]/);
}
```

As an explanation of the pattern, here it is again, this time expressed using Perl's `/.../x` pattern notation that allows embedded comments:

```
return ($str =~ /
            ^       # match beginning of string
            [^@]+   # match non-empty sequence of non-@ characters
            @       # match literal @ character
            [^.]+   # match non-empty sequence of non-period characters
            \.      # match literal period
            [^.]    # require at least one more non-period character
    /x);
```

The pattern match passes correct addresses and flunks several forms of malformed addresses:

```
fred@user-surly.com                okay
fred@central.user-surly.com        okay
fred.rumblebuffin@user-surly.com   okay
fred@user-surly                    bad; domain name has only one component
fred@                              bad; no domain name
@user-surly.com                    bad; no user name
@                                  bad; no user name or domain name
```

`looks_like_email()` is the kind of routine that's likely to be useful in several scripts, so I put it in `WebDB.pm` to make it easily accessible. Putting the address-checking test in a function also is beneficial if you decide at some point that you prefer to use a different pattern for checking addresses—perhaps one that's stricter. Just modify `looks_like_email()` and all your scripts that use it immediately become more strict in what they consider a valid address. You don't need to modify each script individually.[9]

Telephone numbers are one of those types of information that can be specified different ways, so we strip non-digit characters to convert the value to a standard format before storing it. Also, it's a good idea to check the result to make sure it contains at least seven digits, because anything shorter is sure to be malformed:

```
$telephone = WebDB::strip_non_digits ($telephone);
if (length ($telephone) < 7)
{
    push (@errors, "Telephone number must contain at least seven digits");
}
```

If the `@errors` array is non-empty after the field-specific tests, one or more problems were found, so `process_form()` displays the error messages and shows the form again so that the user can correct the problems. Otherwise, everything looks okay and the entry can be added to the contestant table. The query uses `INSERT IGNORE` rather than just `INSERT`. The `IGNORE` keyword is a MySQL-specific extension to the `INSERT` statement that's very helpful in this case. It causes the new entry to be discarded silently if it contains the same telephone number as an existing record entered on the same day. Without `IGNORE`, the script would abort with an error if the entry happens to duplicate an existing record.

An alternative to `INSERT IGNORE` is `REPLACE` (another MySQL extension), which would kick out any earlier entry that matches the new one. However, `INSERT IGNORE` seems a bit fairer in this case. It keeps the existing record, which gives precedence to the earliest entry.

`process_form()` could test whether the new entry was inserted or ignored, by checking the row count returned by the `do()` call. If the result is zero, the record was a duplicate and was discarded. However, we don't bother checking. What for? If someone really is attempting to submit multiple entries, why provide any feedback about the success or failure of the attempt? Letting such users know they failed only serves to alert them that they need to try harder. The principle here is to provide as little information as possible to the bad guys.

9. One caveat: Remember that in a `mod_perl` environment, you'd need to restart Apache to get it to notice that the library file containing the function has been changed.

Picking Winning Entries

It's necessary to pick a winning entry for each contest period. Our giveaway is a daily one, so picking an entry is a matter of selecting a random row from the entries submitted on a given day. For example, the contest for January 17, 2001 closes at the end of the day, so as of January 18, 2001 on, we can pick a winner any time, like this:

```
mysql> SELECT * FROM contestant WHERE entry_date = '2001-01-17'
    -> ORDER BY RAND() LIMIT 1;
+----+------------+-------------+----------------+-----------+
| id | entry_date | name        | email          | telephone |
+----+------------+-------------+----------------+-----------+
| 97 | 2001-01-17 | Paul DuBois | paul@snake.net | 5551212   |
+----+------------+-------------+----------------+-----------+
```

The WHERE clause in the query identifies the candidate rows from which you want to make a selection, ORDER BY RAND() "sorts" the rows into a random order, and LIMIT 1 restricts the result set to the first of these rows. The effect is to pick one of the given day's rows at random.

You cannot use RAND() in an ORDER BY clause prior to MySQL 3.23.2. However, there is a workaround available for older versions. It involves selecting an additional column containing random numbers and sorting the result on that column. This randomizes the result set with respect to the other columns:

```
mysql> SELECT *, id*0+RAND() AS rand_num
    -> FROM contestant WHERE entry_date = '2001-01-17'
    -> ORDER BY rand_num LIMIT 1;
+----+------------+-------------+----------------+------------+--------------+
| id | entry_date | name        | email          | telephone  | rand_num     |
+----+------------+-------------+----------------+------------+--------------+
| 53 | 2001-01-17 | Paul DuBois | paul@snake.net | 6085551212 | 0.2591492526 |
+----+------------+-------------+----------------+------------+--------------+
```

The inclusion of id*0 in the expression defeats the query optimizer, which would otherwise think that the additional rand_num column contains constant values and "optimizes" the ORDER BY rand_num clause by eliminating it from the query.

Now that we have a query for selecting a random row, we can begin picking winning entries. To do so manually, you can run one of the SELECT queries just shown, substituting the appropriate date. Or the computer can do the work automatically if you use a cron job that runs each day to choose a winner from among the preceding day's entries. For example, I might put the following entry in my crontab file to run a script named pick_winner.pl at 5 a.m. every day:

```
0 5 * * * /u/paul/bin/pick_winner.pl
```

Assuming that cron is of the variety that automatically mails job output to the crontab file owner, pick_winner.pl can select a winning entry and display its contents using normal print statements. We can test the script easily by running it from the command line, yet receive the results by mail when the script runs under cron.

`pick_winner.pl` begins by connecting to the database (not shown), and then determines whether there are any relevant entries for the contest date:

```
# default date is "yesterday"
my $contest_date = "DATE_SUB(CURRENT_DATE,INTERVAL 1 DAY)";
# select date in CCYY-MM-DD form
my $date = $dbh->selectrow_array ("SELECT $contest_date");
my $count = $dbh->selectrow_array (
                "SELECT COUNT(*) FROM contestant
                WHERE entry_date = $contest_date");
print "Giveaway contest results for $date:\n";
print "There were $count entries.\n";
```

The `DATE_SUB()` expression evaluates to the date for "yesterday" no matter what the current date is, so we don't need to figure it out ourselves. However, the output from `pick_winner.pl` should indicate which date it's picking a winner for, and printing a `DATE_SUB()` expression literally isn't very useful for that, to say the least. The first `SELECT` query solves this problem.[10] MySQL evaluates `DATE_SUB()` and returns the resulting date in standard CCYY-MM-DD format that will be more meaningful for display purposes. (You could put the `DATE_SUB()` expression inside a call to `DATE_FORMAT()` if you want a different output format, of course.)

Next, `pick_winner.pl` counts the number of entries for the given date and reports the date and entry count. If there are any entries, it picks one at random and displays its contents:

```
if ($count > 0)     # don't bother picking winner if there are no entries
{
    my ($id, $name, $email, $telephone) = $dbh->selectrow_array (
                "SELECT id, name, email, telephone FROM contestant
                WHERE entry_date = $contest_date
                ORDER BY RAND() LIMIT 1");
    # Paranoia check; this shouldn't happen
    die "Error, couldn't select winning entry: $DBI::errstr\n"
                                    unless defined ($id);
    print "The winning entry is:\n";
    print "id: $id\n";
    print "name: $name\n";
    print "email: $email\n";
    print "telephone: $telephone\n";
}
```

Run the script from the command line to make sure it works:

```
% /u/paul/bin/pick_winner.pl
Giveaway contest results for 2001-01-17:
There were 4 entries.
The winning entry is:
id: 397
name: Wendell Treestump
email: wt@stumpnet.org
telephone: 1234567
```

10. Some database engines require that a `SELECT` statement always use at least one table. MySQL does not, so we can use `SELECT` as a calculator that evaluates expressions without reference to any table.

If you have more than one entry, you can run the script several times to verify that it doesn't always pick the same entry.

To make `pick_winner.pl` more useful, we can modify it slightly to accept an optional date from the command line. That way we can override the default date and pick winners for any day. (And, as it happens, by specifying today's date, we can use `pick_winner.pl` to see how many entries have been submitted for the current date so far. In this case we ignore the "winner" entry, of course, because other entries might still arrive.) The modification is quite trivial. Following the line that sets the default date, add another line that checks the `@ARGV` array for a command-line argument:

```
# default date is "yesterday"
my $contest_date = "DATE_SUB(CURRENT_DATE,INTERVAL 1 DAY)";
$contest_date = "'" . shift (@ARGV) . "'" if @ARGV;
```

Now we can invoke the script with an explicit date:

```
% /u/paul/bin/pick_winner.pl 2001-1-15
```

With a little more work, we can have `pick_winner.pl` recognize "dates" such as *`-n`* to mean "*n* days ago." Add the following lines after the `@ARGV`-checking line:

```
if ($contest_date =~ /^'-(\d+)'$/)  # convert -n to n days ago
{
    $contest_date = "DATE_SUB(CURRENT_DATE,INTERVAL $1 DAY)";
}
```

That makes it easier to specify dates on the command line. To pick a winner for three days ago or to see how many entries have been submitted today, use the following commands:

```
% /u/paul/bin/pick_winner.pl -3
% /u/paul/bin/pick_winner.pl -0
```

We attempted to eliminate exact duplicates at record-entry time by standardizing the telephone number that forms part of the unique table index. If you like, you can also look for duplicates at winner-picking time by modifying `pick_winner.pl` to produce additional information that may be helpful for assessing whether your winner is legitimate or fraudulent. To do this, run some queries that look for near matches to the winning entry. If these queries produce any output, you can investigate further. (These results should be considered advisory only; you should examine them manually to evaluate for yourself whether you think something funny is going on.)

One approximate-matching technique can be used to locate questionable entries based on telephone numbers. The number is supposed to be unique among all entries for a given day, but we can eliminate duplicates at entry-submission time only if the number is *exactly* the same as the number in an existing entry. How about the following entries; do they look like duplicates to you?

```
+------+------------+----------------+---------------------+-------------+
| id   | entry_date | name           | email               | telephone   |
+------+------------+----------------+---------------------+-------------+
|  407 | 2001-01-17 | Bill Beaneater | bill@beaneater.net  | 5551212     |
|  413 | 2001-01-17 | Bill Beaneater | bill@beaneater.net  | 6085551212  |
|  498 | 2001-01-17 | Bill Beaneater | bill@beaneater.net  | 16085551212 |
|  507 | 2001-01-17 | Fred Franklin  | fred@mint.gov       | 4145551212  |
+------+------------+----------------+---------------------+-------------+
```

The telephone numbers in the second and third entries aren't exact matches to the number in the first entry, so our `giveaway.pl` script didn't reject them. But as humans, we can easily see that they represent the same telephone number (assuming that 608 is the local area code). The contestant just used the country and area codes to construct phone numbers that are valid but not exactly the same. We can also see that the fourth entry comes from a different area code, so it's not a duplicate—which illustrates why we need to examine approximate-match query output manually.

To find entries with a telephone number that is close to the number in the winning entry, use the following query (`$id` represents the `id` value of the winning entry):

```
SELECT contestant.*
FROM contestant, contestant AS winner
WHERE winner.id = $id
AND contestant.entry_date = winner.entry_date
AND contestant.id != winner.id
AND RIGHT(contestant.telephone,7) = RIGHT(winner.telephone,7)
```

The query involves a self-join. It locates the winning entry in the `contestant` table (under the alias `winner`), and then compares that entry to all others submitted on the same day, looking for a match on the rightmost seven digits of the telephone number. (As with all self-joins, we must refer to the table two different ways to make it clear to MySQL which instance of the table we mean at each point in the query.)

Use the `name` column instead if you want to see whether the winner submitted several entries under the same name. (Perhaps the winner has several phone numbers and submitted one entry per number.) The query is similar to the preceding one except that it looks for matches in the `name` value:

```
SELECT contestant.*
FROM contestant, contestant AS winner
WHERE winner.id = $id
AND contestant.entry_date = winner.entry_date
AND contestant.id != winner.id
AND contestant.name = winner.name
```

The query to look for duplicates on the email address is the same except that both references to the `name` column in the last line should be changed to `email`.

If you decide that a contest winner should be disqualified, just rerun `pick_winner.pl` manually to choose another winner for the appropriate date. (This was in fact one of the reasons for writing it to take an optional date on the command line: That enables you to run the script whenever you want and tell it the day for which you want it to pick a winner.)

Other Uses for Near-Match Detection

The techniques discussed for finding similar records in the `contestant` table can be adapted for other contexts. Suppose you use your database to generate mailing labels from a list of newsletter subscribers or people to whom you send advertising flyers. If you want to cut down on postage costs, you might try modifying these queries to find instances where you're sending multiple mailings to the same person.

Other more general queries can be used to help you assess the extent to which duplicates are present over the entire table (not just for the winning entry). The following queries attempt to identify duplicates for each day, based on the three contestant-supplied values:

```
SELECT entry_date, name, COUNT(*) AS count
FROM contestant
GROUP BY entry_date, name HAVING count > 1

SELECT entry_date, email, COUNT(*) AS count
FROM contestant
GROUP BY entry_date, email HAVING count > 1

SELECT entry_date, RIGHT(telephone,7) AS phone, COUNT(*) AS count
FROM contestant
GROUP BY entry_date, phone HAVING count > 1
```

Notifying the Winner

If you want to contact a contest winner by telephone, you can do that by looking at the phone number in the winner's `contestant` table entry. Of course, that doesn't involve any programming, so it's not very interesting! Let's assume, therefore, that you want to issue the "you're a winner!" notification by email, using the `email` value in the entry. We'll write a `notify_winner.pl` script that uses the `Mail::Sendmail` module discussed earlier in the chapter ("Using Email from Within Applications"). The only piece of information this script needs from us is the winning entry ID number. `notify_winner.pl` can look up the appropriate record and determine from its contents where the message should be sent:

```
#! /usr/bin/perl -w
# notify_winner.pl - Notify giveaway contest winner, given winning entry ID

use strict;
use lib qw(/usr/local/apache/lib/perl);
use Mail::Sendmail;
use WebDB;

# Make sure there's a command-line argument and that it's an integer
@ARGV or die "Usage: $0 winning_entry_id\n";
my $id = shift (@ARGV);
$id =~ /^\d+$/ or die "Entry ID $id is not an integer.\n";
```

continues

continued

```
# Retrieve winning entry from database
my $dbh = WebDB::connect();
my ($entry_date, $name, $email, $telephone) = $dbh->selectrow_array (
                "SELECT entry_date, name, email, telephone FROM contestant
                WHERE id = ?",
                undef, $id);
$dbh->disconnect ();
die "Sorry, there's no entry number $id\n" unless defined ($entry_date);

# Construct mail message and send it (use login name of the user who's
# running the script for the From: address)
my $login = getpwuid ($>) or die "Cannot determine your user name: $!\n";
my %mail =
(
    To => $email,
    From => "$login\@localhost",
    Subject => "Congratulations, you're a winner!",
    Message => "
Congratulations, $name!  You are the winner of our giveaway
contest for the date $entry_date.  To claim your fabulous prize,
please follow these instructions:

<insert instructions for claiming fabulous prize here>."
);

sendmail (%mail) or die "Attempt to send mail failed\n";

exit (0);
```

The script constructs the `From:` address using the login name of the person running it. You might want to change that. (You should also modify the message body to provide appropriate instructions for claiming the prize.)

Dealing with Old Entries

Eventually the `contestant` table may grow quite large. That might not bother you if you want to maintain the entries for statistical purposes. For example, you can run queries such as the following ones to generate a summary of submission activity to assess how popular the giveaway is. The first produces a daily summary, and the second summarizes by month:

```
SELECT
    entry_date,
    COUNT(*) AS count
FROM contestant GROUP BY entry_date

SELECT
    YEAR(entry_date) AS year,
    MONTH(entry_date) AS month,
    COUNT(*) AS count
FROM contestant GROUP BY year, month
```

The `webdb` distribution accompanying the book includes a script `giveaway_summary.pl` that you can install in your Web server's script directory and invoke to display the result of these queries in a browser window.

You may prefer to delete rather than retain old entries. The following query clobbers all submissions from before January 1, 2001:

```
mysql> DELETE FROM contestant WHERE entry_date < '2001-01-01';
```

It's likely you'd want this kind of query to be run automatically so that you don't have to remember to do it yourself. Here's a script, `expire_contestant.pl`, that does so. (To retain old entries for a different number of days, change the `$days` value):

```
#! /usr/bin/perl -w
# expire_contestant.pl - clobber old contestant entries

use strict;
use lib qw(/usr/local/apache/lib/perl);
use WebDB;

# how long to keep entries (in days)
my $days = 30;

my $dbh = WebDB::connect();
$dbh->do (
        "DELETE FROM contestant
        WHERE entry_date < DATE_SUB(CURRENT_DATE,INTERVAL ? DAY)",
        undef, $days);
$dbh->disconnect ();

exit (0);
```

Now, set up an entry in your `crontab` file to run `expire_contestant.pl` as a `cron` job. The format of the entry depends on how often you want expiration to occur. The following entries all run the job at 4 a.m., but vary in the frequency of expiration—choose the form you prefer. The first runs daily, the second each Sunday, and the third on the first day of each month:

```
0 4 * * * /u/paul/bin/expire_contestant.pl
0 4 * * 0 /u/paul/bin/expire_contestant.pl
0 4 1 * * /u/paul/bin/expire_contestant.pl
```

Suggested Modifications

The `contestant` table contains no information indicating who the winners are, so you'll have to remember whom you pick each day. Add a column to the table and modify `notify_winner.pl` to update the appropriate entry by marking it as a winner.

Consider what modifications you'd need to make to change the drawing period from daily to weekly or monthly. Changing the period of time over which contestants are allowed a single entry affects several things:

- What you'd use for the unique key in the contestant table
- How often you draw entries, and the date range you use to select candidate rows
- The notification message in `notify_winner.pl`
- Any approximate-match duplicate detection queries you may be using
- `cron` job scheduling for `pick_winner.pl` and `expire_contestant.pl`

In general, these changes will follow from any changes you make to the `contestant` table to determine the unique key. If you want to conduct a monthly giveaway, for example, you could add `year` and `month` columns to the `contestant` table, and change the unique index from `entry_date` and `telephone` to `year`, `month`, and `telephone`. Then, at entry submission time, use `YEAR(CURRENT_DATE)` and `MONTH(CURRENT_DATE)` to insert the current year and month values. That way, you can continue to use `INSERT IGNORE` to discard duplicate records automatically at entry submission time. Other queries later on in the contest process would select records based on year and month rather than `entry_date`.

Use MySQL to provide you with information about how your Web site is being used and to help you evaluate its effectiveness: Log apparent attempts at submitting duplicate entries. (Check whether the row count returned by the `INSERT IGNORE` query in `giveaway.pl` is zero, and log the entry if so.) This gives you information that might indicate widespread abuse attempts and that can help you decide to implement stricter entry requirements, or perhaps just to terminate the contest altogether.

The `INSERT IGNORE` statement in `giveaway.pl` uses the MySQL-specific `IGNORE` extension. If you want to make the script more portable so that it runs under other databases, yet doesn't die with an error on attempts to enter duplicate entries, how do you do it?

Conducting a Poll

In this section, we'll take a look at polling, a common activity on the Web:

- News organizations often conduct polls on current events: Which candidate do you plan to vote for? How would you assess the president's handling of foreign policy? Is the economy in good shape?

- Sports sites ask for predictions: Who will win the World Series? the Super Bowl? the World Cup? Who's the best athlete of the last decade? Do you think the designated hitter rule should be eliminated?
- Polls enable you to gather feedback about your site or your organization: How easy was it to find what you needed on our Web site? How would you grade our customer service? Do you think our news coverage is objective or biased?
- Ratings are a natural use for polls: How would you rate this restaurant? How much did you enjoy this movie?

Most polls have a fairly standard format: Pose a question to the user and present a set of answers from which to choose, along with a submission button for casting the vote. Poll applications also commonly show the user the current results after a vote is cast. Sometimes a link is provided on the voting form that goes directly to the results page, for users who prefer not to vote or who have voted in the past and just want to see where the vote stands now.

As a simple poll, we'll ask people to vote for their favorite Groundhog's Day celebrity using a script `groundhog.pl`. The two prime candidates are Jimmy the Groundhog in Sun Prairie, Wisconsin, and Punxsutawney Phil in Punxsutawney, Pennsylvania. (This is about the most basic poll you can have—there are only two choices.) The poll script will handle the following operations:

- When first invoked, the script displays a page containing a form that presents the candidate groundhogs using a set of radio buttons, and a button for casting the vote. The page also includes a "see current results" link, in case the user wants to see the vote totals without actually submitting a vote.
- After the user submits a vote, the script adds it to the lucky groundhog's tally, thanks the user for voting, and displays the current results.
- If the user selects the "current results" link rather than casting a vote, the script just displays the results.

The main logic for the `groundhog.pl` script is as follows:

```
my %groundhog_map =     # hash that maps groundhog names to labels
(
    "jimmy" => "Jimmy the groundhog (Sun Prairie, Wisconsin)",
    "phil" => "Punxsutawney Phil (Punxsutawney, Pennsylvania)"
);

print header (),
    start_html (-title => "Vote for Your Favorite Groundhog",
                -bgcolor => "white");

# Dispatch to proper action based on user selection

my $choice = lc (param ("choice")); # get choice, lowercased
```

continues

continued

```
if ($choice eq "")                  # initial script invocation
{
    display_poll_form (\%groundhog_map);
}
elsif ($choice eq "submit")         # tally vote, show current results
{
    process_vote (\%groundhog_map, 1, param ("name"));
}
elsif ($choice eq "results")        # just show current results
{
    process_vote (\%groundhog_map, 0);
}
else
{
    print p (escapeHTML ("Logic error, unknown choice: $choice"));
}

print end_html ();
```

In a sense, this part of the script is similar to the corresponding parts of the `prod_reg.pl` and `giveaway.pl` scripts: It encodes information about the form in a data structure that it passes to the form-generation and form-processing functions. However, the data structure is different because we're going to present a set of radio buttons, not a set of text-input fields. Accordingly, we set up a hash map that associates groundhog names with descriptive labels. We use the names as the values of our radio buttons on the polling form and in the database table used to store the votes. The descriptive labels are more meaningful for users and are used for display in the polling form and results pages.

The map gets passed to each function that needs it (`display_poll_form()` and `process_vote()`). The `process_vote()` routine has a dual purpose; it handles the cases when the user submits a vote or just selects the "see current results" link. The second argument indicates whether a vote is expected; if so, a third argument contains the selected groundhog name.

`display_poll_form()` presents a form containing a radio button for each groundhog and a Submit button:

```
sub display_poll_form
{
my $map_ref = shift;    # groundhog name/label map

    print start_form (-action => url ()),
        p ("Which groundhog is your favorite?"),    # pose question
        radio_group (-name => "name",
                    -values => [ sort (keys (%{$map_ref})) ],
                    -default => "[NO DEFAULT]",
                    -override => 1,
                    -labels => $map_ref,
                    -linebreak => 1),   # display buttons vertically
```

```
            br (),
            submit (-name => "choice", -value => "Submit"),
            end_form ();
        # add link allowing user to see current results without voting
        print hr (),
            a ({-href => url () . "?choice=results"}, "See current results");
    }
```

`$map_ref` refers to the hash that associates the groundhog names with their descriptive labels. This hash can be used as is to supply the `labels` parameter to the `radio_group()` function, but we also need to extract the names for the `values` parameter. However, `values` requires a reference to an array, not an array, so we can't pass the list of names directly:

```
-values => sort (keys (%{$map_ref}))       # incorrect
```

Putting the list inside `[ ]` creates a reference:

```
-values => [ sort (keys (%{$map_ref})) ]   # correct
```

The `default` value for the radio buttons is chosen explicitly not to be equal to either of the groundhog names. This causes the form to be displayed with no button selected, to avoid swaying the vote. Setting the `linebreak` parameter to non-zero causes the buttons to display vertically.

Of course, we're jumping ahead of ourselves here a little bit. What do we do with a vote when it's submitted? We need a table in which to store the vote counts for each candidate. Here's a simple table to hold groundhog names and vote counters:

```
CREATE TABLE groundhog
(
    name    CHAR(10) NOT NULL,              /* groundhog name */
    tally   INT UNSIGNED NOT NULL DEFAULT 0 /* number of votes */
)
```

Initializing the table is trivial; all we need is an `INSERT` statement that adds rows naming each groundhog (we need not set the `tally` column explicitly because its default value is zero):

```
INSERT INTO groundhog (name) VALUES ('jimmy'), ('phil')
```

Now that we have a table, we can use it to tally votes and display results. The `process_vote()` function takes care of this. Its `$tally_vote` argument indicates whether to tally a vote or just display results. However, we need to check whether a vote actually was submitted even if `$tally_vote` is non-zero. (The voting form comes up with no radio button selected; if the user just selects the Submit button, the form's `name` parameter that contains the name of the selected groundhog will be empty.) Therefore, we update the current tally for the appropriate groundhog and thank the user for voting only if we find a legal vote. Then we display the current results:

```
sub process_vote
{
my ($map_ref, $tally_vote, $name) = @_;
my ($dbh, $rs_ref, $row_ref, $sum, @table_row);

    $dbh = WebDB::connect ();
    if ($tally_vote)
    {
        # make sure name was given and that it's one of the legal names
        if (defined ($name) && defined ($map_ref->{$name}))
        {
            $dbh->do ("UPDATE groundhog SET tally = tally + 1 WHERE name = ?",
                        undef, $name);
            print p ("Thank you for voting!");
        }
        else
        {
            print p ("No vote was cast; did you make a choice?");
        }
    }

    print p (" The current results are:");
    # retrieve result set as a reference to a matrix of names and tallies
    $rs_ref = $dbh->selectall_arrayref ("SELECT name, tally FROM groundhog");
    $dbh->disconnect ();

    # compute sum of vote tallies
    $sum = 0;
    map { $sum += $_->[1] } @{$rs_ref};
    if ($sum == 0)  # no results!
    {
        print p ("No votes have been cast yet");
        return;
    }

    # Construct table of results: header line first, then contents.
    # For each groundhog, show votes as a tally and as a percentage
    # of the total number of votes.  Right-justify numeric values.
    push (@table_row, Tr (th ("Groundhog"), th ("Votes"), th ("Percent")));
    foreach $row_ref (@{$rs_ref})
    {
        my $label = $map_ref->{$row_ref->[0]};  # map name to descriptive label
        my $percent = sprintf ("%d%%", (100 * $row_ref->[1]) / $sum);
        push (@table_row, Tr (
                td (escapeHTML ($label)),
                td ({-align => "right"}, $row_ref->[1]),    # tally
                td ({-align => "right"}, $percent)          # % of total
            ));
    }
    print table (@table_row);
}
```

To display the current vote totals, we run a query to retrieve the names and vote tallies from the groundhog table. `selectall_arrayref()` returns a result set as a reference to a matrix (specifically, as a reference to an array, each element of which is a reference to an array containing one row from the table).[11] After retrieving the result set, we sum the `tally` values to determine the total number of votes cast. This has two purposes. First, we can tell from the total whether any votes have been cast yet. Second, the total allows us to calculate the percentage of votes each groundhog has received when we generate the rows of the vote display table. The name-to-label map gives us the descriptive label from the groundhog name, which is HTML-encoded using `escapeHTML()` in case it contains any special characters. (In fact, neither of the labels do, but this approach prevents surprises if we decide to change the labels at a later date.) No encoding is needed for the tally or the percentage values because we know they're numeric and therefore contain no special characters.

Suggested Modifications

Our poll just counts votes. It doesn't tell you when votes were cast. Modify the application to log each vote and when it occurred so that you can perform time-based analysis of poll activity. Write some summary queries that show the number of votes cast each day (week, month, and so on) that your poll is open.

Suppose you hear about another famous groundhog that lives in the city of Bangor, Maine. Consider what you'd need to do to change the set of candidates presented by the poll:

- The current vote counters should be set back to zero to eliminate any head start by the existing candidates.
- You'd need another row in the `groundhog` table for the new candidate.
- You'd need to add another entry to the hash map that associates ground hog names and descriptive labels.

That's not actually too much work. But now suppose you want to conduct a second poll. The `groundhog.pl` script is adequate if groundhog voting is the only poll you'll ever run, but it has some shortcomings for multiple poll presentations. Using our present poll implementation, you'd need to write a script specifically for each poll you want to carry out. You'd also need a separate vote-tallying table for each one. These problems arise because the script is intimately tied to knowledge about this particular poll. It knows precisely what the choices are, what labels should be associated with the names, and the title for the poll form. In Chapter 6, we'll consider a more general polling implementation that eliminates these shortcomings. It's more work to implement, but more flexible.

11. The `selectall_arrayref()` function is useful here to get the entire result set into a data structure because we need to iterate through the result set twice. If we used a row-at-a-time fetch loop, we'd have to run the query twice to process the results twice.

Storing and Retrieving Images

Images are used in many Web applications. This section describes a couple of small scripts that provide you with the ability to load images into MySQL over the Web or from the command line. It also discusses a script that serves images over the Web by pulling them from the database. My reason for placing a section on image storage and retrieval at this point in the chapter is that we'll need image-serving capability for the next section, which demonstrates how to write an electronic greeting card application. One feature of that application is that it enables users to select a picture to display with the card when the recipient views it.

Obviously, to implement that capability, we'll need to be able to send images to the client's browser. And we'll need to supply the application with some images first so that it has something to send! So, here we are.

Images Aren't Special! (Part I)

Although the scripts presented in this section show how to perform storage and retrieval using images, the techniques can be adapted easily for working with any kind of data, not just images. The information here can help you construct a database of sound or video clips, PDF files, compressed data, and so forth.

For storing the images, we'll use a table called `image` that contains a descriptive name for the image, the image itself, a thumbnail (small version) of the image, and the image MIME type:

```
CREATE TABLE image
(
    name        VARCHAR(60) NOT NULL,   # descriptive name for image
    UNIQUE (name),
    image       BLOB NOT NULL,          # image data
    thumbnail   BLOB NOT NULL,          # thumbnail data
    mime_type   VARCHAR(20) NOT NULL    # image MIME type
)
```

The scripts we'll develop refer to images by name. The `name` column has a unique index so that we don't give the same name to two different images. `image` and `thumbnail` are `BLOB` columns, the usual type for binary data. (`VARCHAR` isn't really suitable for such data, because it has a maximum length of just 255 characters.) The `mime_type` column contains values that identify the image format. These will be values such as `image/gif`, `image/jpeg`, `image/png`, and so forth. The image type value applies to both the image and its thumbnail.

To populate the `image` table, we'll write an image-loading script (`upload_image.pl`) that allows images stored on your local disk to be uploaded using your browser for storage into MySQL. It works as follows:

- The initial page presents a form containing a file-upload field for specifying the image file you want to transfer and a field for giving the image a descriptive name. If you have an image of the U.S. flag in a file named `us_flag.jpg`, for example, you might give it a descriptive name of "Stars & Stripes."

- When you submit the form, your browser will send the field values back to the Web server, including the contents of the image file. `upload_image.pl` receives this information, creates a thumbnail version of the image, and stores everything into the database. It also presents a confirmation page that reports the status of the upload operation and displays the image so that you can see that it really was received properly.

Where Should Images Be Stored?

One of the ongoing debates about images and databases is whether to store images in the database or whether to store them in the file system and store only the pathname in the database. `upload_image.pl` shows how to store images in the database so that you'll know how to do it if you want to, but I'm not going to try to settle the debate. If you want more information about the pros and cons of each approach, search the MySQL mailing list archives.

The dispatch logic for `upload_image.pl` is similar to that of several previous applications, so I won't show it here. It invokes `display_upload_form()` to present the entry form and `process_form()` to handle submitted images.

`display_upload_form()` generates the image-selection form. The important thing you should notice about the code is that it uses `start_multipart_form()` rather than `start_form()`. File uploads require form contents to be encoded differently from "regular" forms (otherwise the file content transfer won't work properly):

```
sub display_upload_form
{
    print start_multipart_form (-action => url ()),
            "Image file: ", br (),
            filefield (-name => "image", -size => 60),
            br (),
            "Descriptive name for image: ", br (),
            textfield (-name => "name",
                        -value => "",
                        -override => 1,
                        -size => 60),
            br (), br (),
            submit (-name => "choice", -value => "Submit"),
            end_form ();
}
```

When the user submits an image, `process_form()` is called. This function makes sure the name and image file were both supplied, creates the thumbnail from the image, stores everything in the database, and displays a confirmation page:

```
sub process_form
{
my $name = param ("name");            # image name
my $image = param ("image");          # image file
```

continues

continued

```
my @errors = ();
my $dbh;
my $mime_type;
my ($full, $thumb);
my $serve_url;

    $image = "" unless defined ($image);# convert undef to empty string
    $name = WebDB::trim ($name);        # trim extraneous whitespace from name

    # check for required fields
    push (@errors, "Please supply an image name") if $name eq "";
    push (@errors, "Please specify an image file") if $image eq "";
    if (@errors)
    {
        print p ("The following errors occurred:");
        print ul (li (\@errors));
        print p ("Please click your Browser's Back button to\n"
                . "return to the previous page and correct the problem.");
        return;
    }

    # Form was okay; get image type and contents and create new record.
    # Use REPLACE to clobber any old image with the same name.

    $mime_type = uploadInfo ($image)->{'Content-Type'};
    ($full, $thumb) = read_image_file ($image);
    $dbh = WebDB::connect ();
    $dbh->do (
            "REPLACE INTO image
            (name,image,thumbnail,mime_type)
            VALUES(?,?,?,?)",
                undef,
                $name, $full, $thumb, $mime_type);
    $dbh->disconnect ();

    # Image was stored into database successfully.  Present confirmation
    # page that displays both the full size and thumbnail images.

    print p ("The image upload was successful.");
    # encode the name with escape() for URL, but with escapeHTML() otherwise
    $serve_url = sprintf ("serve_image.pl?name=%s", escape ($name));
    $name = escapeHTML ($name);
    $mime_type = escapeHTML ($mime_type);
    print p ("Image name: $name"),
            p ("MIME type: $mime_type"),
            p ("Full size image:"),
            img ({-src => $serve_url, -alt => $name}), "\n",
            p ("Thumbnail image:"),
            img ({-src => "$serve_url;thumbnail=1", -alt => $name}), "\n";
    # Display link to main page so user can upload another image
    print hr (), a ({-href => url ()}, "Upload next image");
}
```

`process_form()` validates the form by making sure that both the descriptive image name and the file pathname are present. We do the usual thing of trimming extraneous whitespace from the descriptive name. However, we don't do that for the pathname because that value legitimately could begin or end with spaces, and trimming it would change the name. (CGI.pm makes the file contents and information about the file available through the pathname; changing it would render the script unable to access the file.)

If any errors occur during validation, we indicate what they are and instruct the user to return to the preceding page to correct the problems. You may recall that in the discussion of form validation for the product-registration application, I discouraged the approach of having users click the Back button, favoring instead the method of redisplaying the form on the same page as the error messages. Aren't I contradicting that advice here? Yes, I am; and before you read the footnote that explains why, I invite you to consider why this might be.[12]

If the form contents check out okay, we get the image's MIME type using `uploadInfo()`, a CGI.pm function that provides information about the uploaded file, given the filename as an argument. (This function is described in Chapter 4, "Generating and Processing Forms," in the section that discusses the sample form application.) The return value is a reference to a hash of file attributes and values. One of these attributes, `Content-Type`, gives us the image's MIME type.

Next, we read the image from the temporary file in which it is stored. `read_image_file()` (discussed shortly) reads the image file, creates the thumbnail, and returns both values. At this point, we have all the information we need to create a new image table record. The statement that adds the record uses `REPLACE` rather than `INSERT` to make it easy to overwrite an existing image with a new one. (`INSERT` would generate an error, and `INSERT IGNORE` would keep the old image and discard the new one. Neither behavior is desirable here.)

12. The reason I don't follow my own advice here is that CGI.pm won't initialize the value of file-upload fields. This prevents script writers from trying to trick users into uploading specific files, but it also means you can't take advantage of CGI.pm's sticky form behavior for file fields. In fact, this isn't just a CGI.pm behavior; browsers themselves may refuse to honor a `value` attribute for a file-upload field, even if your script includes one by writing the HTML directly. This means we can't properly redisplay the form with the values submitted by the user, and therefore really don't have much choice but to ask the user to return to the previous page.

Images Aren't Special! (Part II)

I have the feeling that I should write a headline in GREAT BIG LETTERS that the `do()` statement used in the `process_form()` function answers the often-asked question, "How, oh how, do I insert images into my database? What's the special trick?" Well, actually, there isn't one. Images are inserted the same way as any other kind of data: Use placeholders or the `quote()` function. The usual thing that gives people trouble putting images in a database is the failure to properly escape the special characters that images usually contain. Images consist of binary data, so attempting to put an image directly into the query string without properly escaping its content almost certainly will fail. If you use a placeholder, or insert the image data into the query string after calling `quote()`, you'll have no problems.

If images seem special compared to text values, due to the need to escape special characters, that's a sign you're probably not processing text properly, either. With text values, you can often get away with not using placeholders or `quote()`, but that doesn't mean it's correct to do so. Text can contain special characters that cause problems, too—such as quote characters. It's important to use placeholders or `quote()` for *all* data, not just images or other binary data. If you do that consistently, you'll likely find that the magic conceptual distinction between text and images disappears.

With the image safely stored away in the database, we can present a confirmation page to the user. Given that we're working with images, we may as well make this a graphical page. Therefore, we'll not only inform the user that the upload succeeded, but we'll also display the image and its thumbnail as well.

The image parts of the confirmation page are nothing more than `<img>` tags that reference an image stored in the database. If the image's name is "My Image," for instance, the tags will look like this:

```
<img src="serve_image.pl?name=My%20Image" alt="My Image">
<img src="serve_image.pl?name=My%20Image;thumbnail=1" alt="My Image">
```

When your browser sees each of these tags in the page, it will send requests to the Web server to retrieve the corresponding images. As the tags show, these requests are handled by another script, `serve_image.pl`. (We have yet to write this script, but we'll get to it soon.) `serve_image.pl` yanks an image out of the database and turns it into a valid image transfer to the browser. The `<img>` tags refer to the script without a leading pathname; we can get away with that if we install `serve_image.pl` in the same directory as `upload_image.pl`. The `name` parameter specifies which image `serve_image.pl` should return to the browser, and the absence or presence of the `thumbnail` parameter indicates whether it should return the full-size image or the thumbnail.

The last thing `process_form()` displays in the confirmation page is a link to the main `upload_image.pl` page so that the user can transfer another image if desired.

We still have to see how to read the contents of the uploaded image file from the temporary file where it's stored and how to produce a thumbnail from it. Let's return to `read_image_file()`, the function that actually does this. This function uses some of the capabilities of `Image::Magick`, a Perl module that allows sophisticated image manipulations to be performed. (You should obtain `Image::Magick` from the CPAN and install it if you don't already have it.)

We pass `read_image_file()` the value of the `image` parameter from the upload form. That parameter contains the name of the file. However, CGI.pm performs a little trick that also allows it to be treated as an open file handle pointing to the uploaded file, so we can use it to read and process the file:

```
use Image::Magick;

sub read_image_file
{
my $fh = shift;             # filename/file handle
my $img = new Image::Magick;
my ($full, $thumb);
my $err;

    # read full-size image directly from upload file
    (read ($fh, $full, (stat ($fh))[7]) == (stat ($fh))[7])
        or error ("Can't read image file: $!");
    # produce thumbnail from full-size image
    $err = $img->BlobToImage ($full);
    error ("Can't convert image data: $err") if $err;
    $err = $img->Scale (geometry => "64x64");
    error ("Can't scale image file: $err") if $err;
    $thumb = $img->ImageToBlob ();
    return ($full, $thumb);
}
```

To handle the image, we create a new `Image::Magick` object, then invoke a few of its methods after reading the contents of the file containing the full-size image. `BlobToImage()` converts the raw image data to a form that `Image::Magick` can use, and `Scale()` resizes the image to produce the thumbnail. The `64x64` argument to `Scale()` does not indicate the final pixel size of the resulting image; it indicates the boundary within which the resized image must fit. (That is, `Scale()` does not change the aspect ratio of the image, only its size.) After scaling the image, we call `ImageToBlob()` to retrieve the thumbnail as a string.[13]

`read_image_file()` uses `error()`, a small utility function that just displays an error message, closes the page, and exits:

```
sub error
{
my $msg = shift;

    print p (escapeHTML ("Error: $msg")), end_html ();
    exit (0);
}
```

13. Here's something that may or may not affect you: I find that `Image::Magick` often crashes in `read_image_file()` at the `ImageToBlob()` call if the image is in GIF format and uses transparency.

Security and File Uploads

File-uploading operations have the potential to cause some security problems. See Chapter 9, "Security and Privacy Issues," for a discussion of these problems and what you might want to do about them.

`upload_image.pl` is complete at this point, so you can install it and try it out right now if you like. Note that although the script should upload images properly, the confirmation page won't yet display the uploaded images. That's because we haven't yet written `serve_image.pl`, the script that handles requests to display images from the `image` table.

Before we create `serve_image.pl`, I want to take a slight detour, because I personally find it really tedious to upload image files one by one over the Web. That's convenient for occasional transfers; but when I have a pile of images, I'd rather transfer them to a UNIX box and run a command-line script that loads them. Here's a command-line equivalent to the `upload_image.pl` script called `load_image.pl`:

```
#! /usr/bin/perl -w
# load_image.pl - load an image file into the image table

use strict;
use lib qw(/usr/local/apache/lib/perl);
use Image::Magick;
use WebDB;

# Determine image file and image name.  Use basename of filename if no image
# name is given.

die "Usage: $0 image_file [ image_name ]\n" unless @ARGV >= 1 && @ARGV <= 2;
my $image_file = shift (@ARGV);
my $image_name = shift (@ARGV);
($image_name = $image_file) =~ s|.*/|| if !defined $image_name;

# determine MIME type of image file from filename extension

my %mime_map = (
    "gif" => "image/gif",
    "jpg" => "image/jpeg",
    "jpeg" => "image/jpeg",
    "jpe" => "image/pjpeg",
    "png" => "image/png"
);
my $mime_type = $mime_map{lc ($1)} if $image_file =~ /\.([^.]+)$/;
die "Cannot determine image MIME type\n" if !defined $mime_type;

# Read image file and generate thumbnail from image

my $img = new Image::Magick;
my ($err, $image_data, $thumbnail_data);
```

```
$err = $img->Read ($image_file);
die "Can't read image file: $err\n" if $err;
$image_data = $img->ImageToBlob ();
$err = $img->Scale (geometry => "64x64");
die "Can't scale image file: $err\n" if $err;
$thumbnail_data = $img->ImageToBlob ();

# Insert new record into the database image table

my $dbh = WebDB::connect ();
$dbh->do (
        "REPLACE INTO image
        (name,image,thumbnail,mime_type)
        VALUES(?,?,?,?)",
            undef,
            $image_name, $image_data, $thumbnail_data, $mime_type);
$dbh->disconnect ();
warn "$image_name loaded\n";     # announce success of image storage operation

exit (0);
```

`load_image.pl` expects to find either one or two arguments on the command line. The first is the image filename. The second, if present, is the descriptive name to give to the image. (If not present, the filename itself is used as the descriptive name.) The script determines the image type from the filename suffix.

Of course, I don't really want to type in a bunch of `load_image.pl` commands at the shell prompt any more than I want to upload images over the Web one by one. So I FTP the images to a directory `Images` on my UNIX box, and then log in there and write a shell script `load_images.sh` that looks like this:

```
#! /bin/sh
./load_image.pl Images/blackcat.jpg "Black Cat"
./load_image.pl Images/flowers.jpg "Flower Bouquet"
etc.
```

The basis of this script can be created using just a few commands:

```
% ls Images > load_images.sh
% chmod +x load_images.sh
% vi load_images.sh
:1,$s/.*/.\/load_image.pl Image\/&/
```

All that needs to be added is the `#!` line at the beginning of the script and the descriptive names at the end of the command lines, and then I can load all the images easily by running `load_images.sh`. This is particularly useful when moving all the images to another machine, because the same script can be used there. (In other words, writing the script creates a repeatable action.)

If you don't have a shell account, you can't use this command-line approach. If you do have one, however, to my mind this method is much preferable to uploading images individually or typing a bunch of individual commands. If you're setting up the `image` table using the `webdb` distribution that accompanies this book, you'll find that its image directory includes the load_images.sh script and an Images subdirectory containing a set of sample images to use.

Serving Images

Now that we can get images into the database, how do we get them out again? This section shows how to write serve_image.pl, the script that retrieves an image from the image table and displays it in a Web page. We need this script so that the confirmation page generated by upload_image.pl can properly show the uploaded images. We'll also use serve_image.pl for image display in the electronic greeting card application developed later in this chapter.

Before we write this script, let's briefly go over the mechanism used to transfer images over the Web to browsers. Images are referenced from Web pages using <img> tags. Typically, the tag refers to a static file on the Web server host. For example, the following tag refers to the "Powered by Apache" image file located in the top directory of the document tree on the host www.snake.net:

```
<img src="http://www.snake.net/apache_pb.gif">
```

A browser retrieves the image by sending the URL named in the src attribute to the Web server. The server in turn satisfies the request by opening the file and sending it to the browser, preceded by some header information that allows the browser to make sense of the data. Typical headers are Content-Type: to specify the MIME type for the image format (image/gif, image/jpeg, and so forth), and Content-Length: to let the browser know how many data bytes to expect.

However, images can be served from sources other than files. If we use a script to duplicate the kind of output the Web server sends when it transfers a static image file, the browser won't care. The script can do this easily by reading a record from the image table and using it to generate a request response. The mime_type column value indicates what kind of Content-Type: header to send and the length of the image data provides a value for the Content-Length: header. We write the headers followed by the image data, and we're done. (Now you see why we store the MIME type in the image table.)

Naturally, we don't want to write a different script for each image, so we'll have serve_image.pl accept a name parameter at the end of the URL specifying the name of the image to display. Additionally, if the URL also includes a thumbnail parameter, we'll serve the thumbnail image rather than the full-size version. And as a final touch, let's give the script the capability to present a gallery page if we invoke it with a gallery parameter rather than an image name. In this case, the script will look up all the image names and descriptions and write an HTML page that includes an <img> tag for the thumbnail version of each one. The thumbnails will be clickable so that you can select any of them to see the corresponding full-size image. (In other words, serve_image.pl will write an HTML page that causes itself to be invoked in its image-serving capacity.) The URLs for invoking serve_image.pl in these various ways look like this:

```
http://www.snake.net/cgi-perl/serve_image.pl?name=image_name
http://www.snake.net/cgi-perl/serve_image.pl?name=image_name;thumbnail=1
http://www.snake.net/cgi-perl/serve_image.pl?gallery=1
```

The dispatch logic for serve_image.pl extracts the URL parameters and determines what to do as follows:

```
if (defined (param ("name")))
{
    display_image (param ("name"), param ("thumbnail"));
}
elsif (defined (param ("gallery")))
{
    display_gallery ()
}
else
{
    error ("Unknown request type");
}
```

The image-serving code really is pretty trivial. It checks whether to use the full-size image or the thumbnail, and then looks up the appropriate record from the image table, determines the image length from the image data, and writes the headers followed by the data:

```
sub display_image
{
my ($name, $show_thumbnail) = @_;
my $col_name = (defined ($show_thumbnail) ? "thumbnail" : "image");
my ($dbh, $mime_type, $data);

    $dbh = WebDB::connect ();
    ($mime_type, $data) = $dbh->selectrow_array (
                    "SELECT mime_type, $col_name FROM image WHERE name = ?",
                    undef, $name);
    $dbh->disconnect ();
    # did we find a record?
    error ("Cannot find image named $name") unless defined ($mime_type);

    print header (-type => $mime_type, -Content_Length => length ($data)),
            $data;
}
```

By default, the header() function writes a Content-Type: header with a value of text/html if you don't specify any type parameter. We need to override that with the MIME type of the image, otherwise the browser may misinterpret the output and try to display the image data as text. (You can see whether your browser makes a mess of images by removing the `type` parameter from the `header()` call and then requesting an image from your browser.)

If the `gallery` parameter is present in the URL, `serve_image.pl` generates an HTML page that displays the thumbnails for all the images in the image table:

```
sub display_gallery
{
my ($dbh, $sth);

     print header (), start_html ("Image Gallery");

    $dbh = WebDB::connect ();
    $sth = $dbh->prepare ("SELECT name FROM image ORDER BY name");
    $sth->execute ();
    # we're fetching a single value (name), so we can call fetchrow_array()
    # in a scalar context to get the value
    while (my $name = $sth->fetchrow_array ())
    {
        # encode the name with escape() for the URL, with escapeHTML() otherwise
        my $url = url () . sprintf ("?name=%s", escape ($name));
        $name = escapeHTML ($name);
        print p ($name),
            a ({-href => $url},     # link for full size image
                # embed thumbnail as the link content to make it clickable
                img ({-src => "$url;thumbnail=1", -alt => $name})
            ),
            "\n";
    }
    $sth->finish ();
    $dbh->disconnect ();

    print end_html ();
}
```

For each image, `display_gallery()` displays the image name and an `<img>` tag for the thumbnail. The `<img>` tag is embedded within a hyperlink that takes the user to the full-size image; you can click any thumbnail to see the larger version.

The `error()` utility routine handles any problems by presenting a short error page. It differs from the version used in `upload_image.pl` slightly because it generates a complete HTML page:

```
sub error
{
my $msg = shift;

    print header (),
            start_html ("Error"),
            p (escapeHTML ($msg)),
            end_html ();
    exit (0);
}
```

Suggested Modifications

`upload_image.pl` doesn't check whether the uploaded file really is an image. Is that a problem? If so, can you fix it?

It's possible to load into the `image` table images that some browsers may be unable to display. For example, older browsers likely won't know what to do with images in PNG (Portable Network Graphics) format. Modify the `display_gallery()` function of `serve_image.pl` to exclude images except those in formats the browser understands.

If you load lots of images into the `image` table, you'd probably want to modify the gallery display code in `serve_image.pl` to split up the gallery into a multiple-page display. Techniques for multiple-page presentations are described in Chapter 7, "Performing Searches."

`serve_image.pl` assumes that it is supposed to read images from the `image` table. If you use it to serve images on behalf of many different applications, you may find it limiting to share the `image` table among them all. Modify `serve_image.pl` to accept a `table` parameter on the URL so that applications can specify which table to use. To preserve compatibility with its original behavior, have the `default` table be `image` if no `table` parameter is present.

Modify `upload_image.pl` and `load_image.pl` to store image files in the file system and reference them from the database by storing the pathname in the `image` table. When you do this, can you toss `serve_image.pl` in the trashcan?

Electronic Greeting Cards—Send a Friend a Greeting

Our next application enables users to construct electronic greeting cards and send them to friends. You use your browser to create a card, and the recipient uses a browser to view it. This is a more ambitious (and complex) undertaking than any of our previous applications because it has to do a whole bunch of stuff:

- When you first visit the application, you see a card information form into which you enter the name and email address for the recipient and for yourself, as well as the text of the greeting you want to send.
- If you want to select a picture to be displayed with the card, the application switches to a page that displays a gallery of images. After you pick one, the application switches back to the original information form. If you decide to select a different picture, the process repeats.
- After you've finished constructing the card, the application assigns it an expiration date and generates an email message to the recipient indicating that a card is waiting and the URL to use for viewing it.

- When the recipient issues a request for the card, the application retrieves the appropriate record from the database, generates a Web page that displays it, and updates the record to indicate that the recipient has seen the card. If you asked to be notified when the recipient views the card, the application sends you an email message to that effect.
- The application removes old cards from the database periodically.

In effect, the application enables one person to communicate asynchronously with another by means of email and the Web, and it uses MySQL to provide the persistent storage that makes this possible. The application consists of several scripts that handle the various tasks involved:

- `make_ecard.pl` manages the card-creation process. It presents the card information form and the picture gallery page.
- When the recipient requests the card, `show_ecard.pl` displays it, updates the record as having been seen, and notifies the card sender.
- We need a supply of images and a means of displaying them to be able to present pictures with cards. `make_ecard.pl` and `show_ecard.pl` rely for this on the `image` table and the `serve_image.pl` script developed in the preceding section, "Storing and Retrieving Images." If you haven't yet read that section, it would be a good idea to do so.
- `expire_ecard.pl` removes old cards from the database. It runs periodically as a `cron` job that checks card expiration dates.

Card construction can take place across the span of several page requests, so we may need to store and retrieve the card to and from the database many times before it's completed. If the user switches from the card information form to the image gallery page, for example, the contents of the form are saved to the database before presenting the gallery. After the user selects a picture, we pull the information back out of the database and use it to initialize the form before redisplaying it. Because we need to tie together multiple page requests (so that we can associate them all with the same card), we'll create a unique ID when a user begins the card-creation process, and carry that ID along at each stage until the card is finished.

The `ecard` table for storing card records looks like this:

```
CREATE TABLE ecard
(
    id              INT UNSIGNED NOT NULL AUTO_INCREMENT,   # card ID number
    PRIMARY KEY (id),
    recip_name      VARCHAR(255),   # recipient name and email address
    recip_email     VARCHAR(255),
    sender_name     VARCHAR(255),   # sender name and email address
    sender_email    VARCHAR(255),
    sender_notify   ENUM('N','Y'),  # notify sender when recipient views card?
    message         TEXT,           # card message
    picture         VARCHAR(40),    # name of picture to show with card
```

```
        expiration      DATE,           # when to expire card
        viewed          DATE            # when recipient first viewed card
    )
```

The `id` column specifies the unique identification number that is used to track each card throughout the card-construction process. We'll also include it as part of the URL that is sent to the recipient so that when a recipient requests a card, we can figure out which one to display.

The recipient name and email address allow us to notify the recipient. (Strictly speaking, notification requires only the email address, but a name allows the application to generate messages that are more personal.)

The sender name and email address are needed so that we can inform the recipient who's responsible for sending the message, and also for generating a notification message if the sender wants to be told when the recipient views the card. `sender_notify` is a two-value column indicating whether the card sender desires this kind of notification.

The `message` and `picture` fields comprise the content of the greeting. `message` contains the text of the greeting, and `picture` indicates which image from the `image` table to display with the message. (A `picture` value of `NULL` indicates "no picture.")

The `ecard` table also contains two `DATE` columns. `expiration` is `NULL` until the sender completes the card, and then it is set to indicate the date when the card can be deleted from the `ecard` table. We'll have the application refuse to modify any card that already has the expiration date set. This convention prevents accidental duplicate card submissions if the card sender clicks the browser's Back button after sending the card and then selects the Send Card button again. This same convention is also a modest security enhancement; it prevents someone else from coming along later and modifying a card that's already been completed. The `viewed` column indicates when the recipient asked to see the card. It's `NULL` until such a request is received.

Many of these columns are mandatory. Before a card can be sent, we'll enforce the constraint that the names and email addresses all must be supplied, as well as the message text. Because a card may be created in several steps, however, the point at which this information is required does not occur until the user finally indicates the card is finished. That means we must enforce the "fields required" rule only when the user selects the Send Card button and not before.

Card Storage and Retrieval Utility Routines

Our card-creation script `make_ecard.pl` needs to perform several types of operations on card records. The script itself represents card information internally as a hash with key names that are the same as columns in the `ecard` table. However, we'll need to exchange that information back and forth with the client's Web browser, and we'll need to store the hash into and retrieve it from the corresponding database record:

- When the card information form is displayed, we look up any information about the card that exists in the database and use it to initialize the form that the user sees.

- When the user switches to the image gallery page or indicates that the card is complete, we extract the contents of the information form using CGI.pm's `param()` function and save the card to the database.

To do all this, we'll use three functions: `extract_card_params()` to get card values from the script environment, `lookup_card_record()` to retrieve a card from the database, and `update_card_record()` to store a card in the database. All three functions require the card's `id` value so they can tell which card to operate on. As we'll see shortly, the main logic of the application makes sure this value is known before any of these utility routines are called.

`extract_card_params()` looks in the script parameter environment for values that the user can specify. This does not include the `expiration` or `viewed` dates because the user has no control over them, and they don't come into play until the card has been completed, anyway. The function extracts the relevant card values from the parameter space, constructs a hash from them, and returns a reference to the hash. We create a slot for each parameter to make sure each one is defined and trim the values to eliminate any extraneous spaces:

```
sub extract_card_params
{
my $card_ref = {};

    $card_ref->{id} = param ("id");
    foreach my $param ("recip_name", "recip_email", "sender_name",
                        "sender_email", "sender_notify", "message", "picture")
    {
        $card_ref->{$param} = WebDB::trim (param ($param));
    }
    return ($card_ref);
}
```

`lookup_card_record()` fetches a card from the database given the card ID number and returns it as a hash reference:

```
sub lookup_card_record
{
my ($dbh, $id) = @_;
my ($sth, $ref);

    $sth = $dbh->prepare ("SELECT * FROM ecard WHERE id = ?");
    $sth->execute ($id);
    $ref = $sth->fetchrow_hashref ();
    $sth->finish ();
    return ($ref);  # undef if card doesn't exist
}
```

To update a card, we shove it back into the database by converting the card hash to an `UPDATE` statement. Again, the ID number identifies the proper card:

```
sub update_card_record
{
my ($dbh, $card_ref) = @_;
my ($stmt, @placeholder);

    # don't store an empty value in this column
    $card_ref->{sender_notify} = "N" if $card_ref->{sender_notify} ne "Y";

    # Construct the SET clause listing the column values to be updated.
    # Skip the id element here (it's used in the WHERE clause, not the
    # SET clause).
    foreach my $key (keys (%{$card_ref}))
    {
        next if $key eq "id";
        $stmt .= "," if $stmt;              # separate assignments by commas
        $stmt .= "$key = ?";                # construct placeholder reference
        push (@placeholder, $card_ref->{$key}); # save placeholder value
    }
    return unless @placeholder; # do nothing if there's nothing to update

    # complete the statement, then execute it
    $stmt = "UPDATE ecard SET $stmt WHERE id = ?";
    push (@placeholder, $card_ref->{id});
    $dbh->do ($stmt, undef, @placeholder);
}
```

Now that we have these support routines in place, we can see how they fit into the overall architecture of the card construction process.

Card Construction Main Logic

When you invoke `make_ecard.pl`, it needs to know whether you're beginning a new card or are already in the middle of creating one. This is accomplished using the card ID number, which exists during the card-making process, but not before. The first part of the script therefore checks for an `id` parameter prior to executing the dispatch code:

```
my $card_ref;                   # card information hashref
my $dbh = WebDB::connect ();    # connect to database

# Determine whether to begin a new card or continue working
# on an existing one by checking for a card ID value.

my $id = param ("id");
if (!defined ($id) || $id !~ /^\d+$/)
{
    # ID is missing (or is not an integer, and is therefore malformed).
    # Create new ecard record; this generates a new ID number
    $dbh->do ("INSERT INTO ecard SET id = NULL");
    $id = $dbh->{mysql_insertid};           # retrieve ID number
    param (-name => "id", -value => $id);   # place ID in parameter space
    $card_ref = extract_card_params ($id);  # construct standard card hash
}
```

continues

continued

```
else
{
    # ID was found, so the card should already exist in the database.
    # Make sure it does and that the expiration date hasn't been set.
    # (If that date has been set, the card has already been sent!)
    $card_ref = lookup_card_record ($dbh, $id);
    if (!$card_ref || $card_ref->{expiration})
    {
        # error - disconnect and close the page; we need proceed no further
        $dbh->disconnect ();
        print p ("No card with ID $id exists, or card has already been sent");
        print end_html ();
        exit (0);
    }
}
```

If `make_ecard.pl` finds no ID number, it begins a new card by creating a new record in the `ecard` table. The `INSERT` statement causes MySQL to create a new ID number; `id` is an `AUTO_INCREMENT` column and setting it to `NULL` generates the next number in the sequence. This number is available as the value of the `mysql_insertid` database handle attribute after executing the `INSERT`. (We could also determine the value by issuing a `SELECT LAST_INSERT_ID()` query, but `mysql_insertid` provides the same information without the overhead of a second query.) Then we put the `id` value into the parameter space and construct a standard hash structure. At this point, all elements of the hash except `id` are empty.

If `make_ecard.pl` does find an `id` value, that means the user is working on an existing card, so there should already be a record for it in the database. We look up the record to make sure it really exists, and then check the expiration date to verify that it hasn't already been set. (The `expiration` value remains `NULL` until the user sends the card; if it's set, that indicates the user is probably accidentally sending the card again or that someone else is attempting to modify it. Either way, we refuse to continue any further.)

After the preceding initial code executes, we know that we have a database record representing the current contents of the card. We may also have new information in the script's parameter space, if the user has just submitted the card information form or chosen an image from the picture gallery. The dispatch code determines what to do, based as usual on the value of the `choice` parameter. This code is structured around the choices the user can make on the form page and the gallery page:

- The information form has two buttons—Select Picture (or Change Picture if an image has already been chosen) and Send Card.
- The gallery page shows a set of images, any of which many be selected. There is also a Continue button if the user decides not to select any picture.

If the value of `choice` is `empty` or `continue`, we populate the card form with whatever information has already been specified and display it. (The value will be `continue` if the user was just viewing the gallery page but decided not to choose a picture.) If `choice` is `select picture` or `change picture`, we switch to the image gallery page. If `choice` is `add_picture`, the user just chose a picture. Finally, if the value of `choice` is `send card`, the user has completed the card and we can save it for good and notify the recipient.

```
my $choice = lc (param ("choice")); # get choice, lowercased

if ($choice eq "" || $choice eq "continue")
{
    # New card or user declined to choose a picture from the
    # gallery page.  Just redisplay the card information form.
    display_entry_form ($card_ref);
}
elsif ($choice eq "select picture" || $choice eq "change picture")
{
    # display image gallery (but save form info first)
    $card_ref = extract_card_params ($id);
    update_card_record ($dbh, $card_ref);
    display_gallery ($dbh, $id);
}
elsif ($choice eq "add_picture")
{
    # User chose a picture from the gallery page. Extract the picture
    # name and add it to the card hash, then redisplay card form.
    $card_ref->{picture} = param ("picture") if param ("picture");
    update_card_record ($dbh, $card_ref);
    display_entry_form ($card_ref);
}
elsif ($choice eq "send card")  # all done; send the card
{
    $card_ref = extract_card_params ($id);
    send_card ($dbh, $card_ref);
}
else
{
    print p (escapeHTML ("Logic error, unknown choice: $choice"));
}
```

Displaying the Card Information Form

The code for displaying the form is a bit different from most of those we've written so far. Each field-generating call is passed a `value` parameter and `override` is turned on so that the value becomes the field's default value. Normally we might rely on CGI.pm's sticky behavior to initialize a form with any previous values. That doesn't work for `make_ecard.pl`, because sometimes the values come from the database rather than the parameter space. (This is the case if the user was just viewing the gallery page, for example.) At any rate, `$card_ref` always points to the card's current contents, whether they come from the database or the parameter space, so we can use it to provide the form's default values.

```
sub display_entry_form
{
my $card_ref = shift;   # reference to card hash

    if ($card_ref->{picture} ne "") # If the card has a picture, display it
    {
        my $img_url = sprintf ("serve_image.pl?name=%s",
                                escape ($card_ref->{picture}));
        print img ({-src => $img_url,
                    -alt => escapeHTML ($card_ref->{picture})});
    }

    print start_form (-action => url ()),
        hidden (-name => "id",
                -value => $card_ref->{id},
                -override => 1),
        hidden (-name => "picture",
                -value => $card_ref->{picture},
                -override => 1),
        p ("Use this form to send an electronic greeting card to a friend."),
        "Person to whom you're sending the card:",
        table (
            Tr (
                td ("Name:") ,
                td (textfield (-name => "recip_name",
                                -value => $card_ref->{recip_name},
                                -override => 1, -size => 60))
            ),
            Tr (
                td ("Email address:"),
                td (textfield (-name => "recip_email",
                                -value => $card_ref->{recip_email},
                                -override => 1, -size => 60)),
            )
        ),
        br (), br (), "Message to send to recipient:",
        br (),
        textarea (-name => "message",
                    -value => $card_ref->{message},
                    -override => 1,
                    -rows => 3,
                    -cols => 60,
                    -wrap => "virtual"),
        br (), br (),
        "Please identify yourself (the person from whom the card is sent):",
        table (
            Tr (
                td ("Name:") ,
                td (textfield (-name => "sender_name",
                                -value => $card_ref->{sender_name},
                                -override => 1, -size => 60))
            ),
            Tr (
                td ("Email address:"),
                td (textfield (-name => "sender_email",
                                -value => $card_ref->{sender_email},
                                -override => 1, -size => 60)),
            )
        ),
        br (),
        p ("Would you like to be notified when the recipient views the card?"),
        # Note: if $card_ref->{sender_notify} is empty, the default
        # becomes the first radio button ("N"), which is what we want.
        radio_group (-name => "sender_notify",
                    -values => [ "N", "Y" ],
                    -labels => { "N" => "No", "Y" => "Yes" },
                    -default => $card_ref->{sender_notify},
                    -override => 1),
        br (), br (),
        submit (-name => "choice",
                -value => ($card_ref->{picture} ne "" ?
                            "Change" : "Select") . " Picture"),
        " ",
        submit (-name => "choice", -value => "Send Card"),
        end_form ();
}
```

The form display code adapts to the presence or absence of a picture selection in two ways. First, if the card has a picture, we display it above the form so that the user can see it. (The code generates an `<img>` tag that invokes our `serve_image.pl` script to obtain the image.) Second, the picture selection button title is Select Picture if no picture has been chosen, and Change Picture otherwise.

The form also includes a couple of hidden fields. We need the `id` value to identify the card. But we also carry along the `picture` value. That's not something the user can specify in the form, but we don't want to lose the value by not including it here. (Otherwise `param("picture")` will be empty when the user submits the form and we update the card record in the database using the information in that form.)

Presenting the Picture Gallery

When the user selects the Select Picture/Change Picture button, `make_ecard.pl` calls `display_gallery()` to present a gallery page that shows thumbnails of the images in the `image` table. This is preferable to displaying the full-size images because thumbnails require less bandwidth to transfer, load more quickly, and result in a more compact display. (Full-size image display is better limited to showing the card itself, which involves only one picture.)

We'll display each thumbnail with its name and make both of them hyperlinks so that the user can click either one to select an image for the card. To reduce the length of the gallery page, we'll arrange the images into a table and present several images per row. Six images per row significantly reduces the amount of vertical scrolling the user must do to see the entire gallery, without making the table so wide that the user likely would need to scroll horizontally to see all the images in a given row. (This is a fairly arbitrary choice, tied to my decision to use 64×64 for the thumbnail size when we built the `image` table. If you wanted to get fancier, you could modify the `image` table to store the dimensions of the thumbnails, and then attempt to determine from those values what a reasonable column count would be for the gallery table.)

What if the user takes a look at the gallery and decides not to select any of them? We could provide instructions to click the Back button, but another way to handle this issue and provide the user a sense of continuing to move forward through the card-creation process is to put a Continue button on the page along with a caption "select Continue to choose no image." Thus, `display_gallery()` presents a page that consists of a table of images followed by a short form containing only the Continue button.

```
sub display_gallery
{
my ($dbh, $id) = @_;
my ($image_list_ref, $nimages, $nrows, $ncols);
my @table_row;

    print start_form (),
        # include card ID so next page knows which card to use
        hidden (-name => "id", -value => $id, -override => 1);

    # Select the names of all images available in the gallery
    $image_list_ref = $dbh->selectcol_arrayref (
                    "SELECT name FROM image ORDER BY name");
    if (!$image_list_ref || @{$image_list_ref} == 0)
    {
        print p ("Sorry, there are no pictures available at this time"),
            submit (-name => "choice", -value => "Continue"),
            end_form ();
        return;
    }
    print p ("To make a picture selection, click on the picture or\n"
            . "on its name. To continue without choosing a picture,\n"
            . "select the Continue button at the end of the page.\n");

    # Determine the number of images available. Then, given the
    # number of columns to display in the table, figure out how
    # many rows there will be.
    $nimages = @{$image_list_ref};
    $ncols = 6;
    $nrows = int (($nimages + $ncols - 1) / $ncols);
    for my $row (0 .. $nrows - 1)
    {
        # construct a string containing the cells in the row
        my @cell;
        for my $col (0 .. $ncols - 1)
        {
            if (($row * $ncols) + $col < $nimages)  # display image in cell
            {
                my $name = $image_list_ref->[$row * $ncols + $col];
                # URL for displaying the image thumbnail
                my $img_url = sprintf ("serve_image.pl?name=%s;thumbnail=1",
                                        escape ($name));
                # URL for selecting this picture and adding it to the card
                my $select_url = url ()
                        . sprintf ("?choice=add_picture;id=%d;picture=%s",
                                    $id, escape ($name));
                # display image name and thumbnail; make each one a hyperlink
                # that adds the picture to the card
                push (@cell,
                        a ({-href => $select_url}, escapeHTML ($name))
                        . br ()
                        . a ({-href => $select_url},
                            img ({-src => $img_url,
                                    -alt => escapeHTML ($name)}))
                        );
            }
            else                                    # display empty cell
            {
                # this happens on last row when there aren't
                # enough images to fill the entire row
                push (@cell, "&nbsp;"); # put non-breaking space in cell
            }
        }
        push (@table_row,
                Tr (td ({-valign => "top", -align => "center"}, \@cell)));
    }
    print table ({-border => 1}, @table_row),
        p ("Select Continue to return to main card form\n"
            . "without making a picture selection.\n"),
        submit (-name => "choice", -value => "Continue"),
        end_form ();
}
```

If the user selects the Continue button, `display_gallery()` posts a form containing a `choice` value of `Continue` and an `id` parameter that identifies the card. (The `id` value is contained in the form as a hidden value to make sure it gets communicated back to `make_ecard.pl`.) The script processes the Continue button by just redisplaying the form to show the current contents of the card.

On the other hand, if the user selects a picture name or thumbnail, each of those is linked to a URL that contains the appropriate parameters for adding the picture to the card. Each URL contains a `choice` value indicating that a picture was chosen, the ID number of the card, and a `picture` parameter indicating which picture to add:

```
make_ecard.pl?choice=add_picture;id=n;picture=name
```

The dispatch code for the `add_picture` choice sets the `picture` attribute of the card hash using the value of the `picture` parameter from the URL. (If there was already a `picture` value in the card hash, it will be replaced by the picture named in the URL. This way we don't lock the user into a given picture.) Then we store the modified card information in the database and redisplay the card information form:

```
elsif ($choice eq "add_picture")
{
    # User chose a picture from the gallery page. Extract the picture
    # name and add it to the card hash, then redisplay card form.
    $card_ref->{picture} = param ("picture") if param ("picture");
    update_card_record ($dbh, $card_ref);
    display_entry_form ($card_ref);
}
```

We're making progress, but we still need to take care of the code to store the final card and send the notification email.

Sending the Card

When the user selects the Send Card button to complete the card, `make_ecard.pl` must take the following steps:

- Extract the card information from the form and make sure all the required fields are present.
- Assign an expiration date. (As a side-effect, this marks the card as "done," a convention we use to prevent double submissions or attempts to tamper with the card.)
- Store the card in the database.
- Send email to the recipient containing instructions for viewing the card.
- Send email to the user who's sending the card noting that it's been sent and also containing instructions for viewing it.
- Display a confirmation page to the user.

Extracting the card from the form is just a matter of calling `extract_card_params()` to construct the card hash. This is done in the main dispatch logic. The other steps are handled by the `send_card()` function:

```
sub send_card
{
my ($dbh, $card_ref) = @_;
my @errors;
my %req_field_map =
(
    "recip_name" => "Recipient's name",
    "recip_email" => "Recipient's email address",
    "message" => "The message to send to the recipient",
    "sender_name" => "Your name",
    "sender_email" => "Your email address"
);
my $card_life = 30;         # how long to retain the card, in days
my ($iso_date, $desc_date);
my ($url, $recip_url, $sender_url);
my %mail;

    # Make sure required fields are filled in
    foreach my $key (keys (%req_field_map))
    {
        # if field is required but missing, it's an error
        if (defined ($req_field_map{$key}) && $card_ref->{$key} eq "")
        {
            push (@errors, $req_field_map{$key} . " must be filled in");
        }
    }

    # Perform additional constraint checking: email fields must look
    # like addresses.

    if ($card_ref->{recip_email} ne ""
        && !WebDB::looks_like_email ($card_ref->{recip_email}))
    {
        push (@errors,
            "Recipient email address is not in user\@host.name format");
    }
    if ($card_ref->{sender_email} ne ""
        && !WebDB::looks_like_email ($card_ref->{sender_email}))
    {
        push (@errors,
            "Your email address is not in user\@host.name format");
    }

    if (@errors)
    {
        print p ("The following problems were found in the card form:");
        print ul (li (\@errors));           # print error messages
```

continues

continued

```
        display_entry_form ($card_ref);     # redisplay form
        return;
    }

    # Get expiration date in ISO and descriptive formats
    ($iso_date, $desc_date) = get_card_expiration ($card_life);
    # Assign expiration date and store final card in database
    $card_ref->{expiration} = $iso_date;
    update_card_record ($dbh, $card_ref);

    # Get full URL of current script and convert last component to name
    # of card-display script. Then add card ID and viewer role parameters.

    $url = url ();
    $url =~ s/[^\/]+$/show_ecard.pl/;
    $recip_url = $url . sprintf ("?id=%d;recip=%s",
                        $card_ref->{id}, escape ($card_ref->{recip_email}));
    $sender_url = $url . sprintf ("?id=%d;sender=%s",
                        $card_ref->{id}, escape ($card_ref->{sender_email}));

     # Send email to card recipient

    $mail{To} = $card_ref->{recip_email};
    $mail{From} = $card_ref->{sender_email};
    $mail{Subject} = "An electronic greeting card for you!";
    $mail{Message} = "
Hello, $card_ref->{sender_name} ($card_ref->{sender_email}) has sent you
an electronic greeting card.

You can view the card with your Web browser at the following address:

$recip_url

The card will be available for $card_life days (until $desc_date).
";
    sendmail (%mail);

    # Send email to card sender

    $mail{To} = $card_ref->{sender_email};
    $mail{From} = $card_ref->{sender_email};
    $mail{Subject} = "Your card to $card_ref->{recip_name}";
    $mail{Message} = "
This message is for your records.  You sent an electronic greeting card to:
$card_ref->{recip_name} ($card_ref->{recip_email})

You can view the card with your Web browser at the following address:

$sender_url
```

```
The card will be available for $card_life days (until $desc_date).
";
    sendmail (%mail);

    # display confirmation page
    print p ("Your card has been sent. Thank you for using this service.");
}
```

The `send_card()` function first validates the form by checking for required fields and making sure the values in the email fields actually look like email addresses. (The test for required fields cannot be done earlier in the card-construction process because we enable the user to leave any of the fields blank up to the point when the card is to be sent.) It's unnecessary to trim whitespace here like we did in the validation procedure for most of the applications developed earlier in the chapter; that already has been done by the `extract_card_params()` function. `looks_like_email()` is one of the utility routines in the `WebDB` module. It runs a pattern test on a string to verify that it looks like a legal email address.

If any errors are found, we show the error messages and redisplay the form so that the user can correct the problems. Otherwise, the card checks out okay, so we assign it an expiration date and update the record in the database:

```
# Get expiration date in ISO and descriptive formats
($iso_date, $desc_date) = get_card_expiration ($card_life);
# Assign expiration date and store final card in database
$card_ref->{expiration} = $iso_date;
update_card_record ($dbh, $card_ref);
```

`get_card_expiration()` calculates the expiration date, given the number of days the card should "live." The date can be obtained from either MySQL or Perl, but however we get it, we'll need it in two formats. The date must be in ISO 8601 format (CCYY-MM-DD) for storage into MySQL. For display in the email messages, we'll use a more descriptive format—for example, January 23, 2001 rather than 2001-01-23.

To get the expiration date from MySQL, we can use `CURRENT_DATE` to get today's date and `DATE_ADD()` to perform date arithmetic. That returns the expiration date in ISO format. To get the descriptive format, we do the same thing but pass the result to `DATE_FORMAT()`. Here's a query to retrieve the expiration date in both formats, where `$card_life` represents how many days to retain the card in the database:

```
($iso_date, $desc_date) = $dbh->selectrow_array (
    "SELECT
        DATE_ADD(CURRENT_DATE,INTERVAL $card_life DAY),
        DATE_FORMAT(DATE_ADD(CURRENT_DATE,INTERVAL $card_life DAY),'%M %e, %Y')");
```

Alternatively, we can get the expiration date from Perl using `time()` and `localtime()`. `time()` returns the current time in seconds. We can add to that value the number of seconds in 30 days, and then pass the result to `localtime()` to convert it to an eight-element array containing the various parts of a date. (Month, day, and year are contained in elements 3 through 5 of the array.) The `year` value represents the number of years relative to 1900, so we add 1900 to get the absolute year. The `month` is a numeric value in the range 0 to 11; we can use it as is to index into an array of month names to get the month name, or add one to get the actual month number. The code looks like this:

```
@monthname = (
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
);
($day, $month, $year) = (localtime (time () + (60*60*24*$card_life)))[3..5];
$year += 1900;      # convert year to 4-digit form
$iso_date = sprintf ("%04d-%02d-%02d", $year, $month+1, $day);
$desc_date = "$monthname[$month] $day, $year";
```

Which method of calculating the expiration date is preferable? Personally, I prefer the `SELECT` version because we can get both `date` values using a single `selectrow_array()` call. But that method incurs the overhead of a round trip over the network to the MySQL server. Getting the date directly from Perl does not, so it's more efficient.

After calculating the expiration date and using it to update the card record in the database, the only thing left to do is send email to the card recipient and the sender. These notification messages can be simple or fancy. On the fancy side, you'll notice when you create a card at one of the big e-card sites is that they include lots of extra stuff in their email, much of it related to advertising. That's the kind of thing you'll have to customize for yourself. For our purposes here, we'll confine the message content to the essentials related only to the card content. The message sent to the recipient will look like this:

To: *recipient*
From: *sender*
Subject: An electronic greeting card for you!
Hello, *sender* has sent you an electronic greeting card.
You can view the card with your Web browser at the following address:
URL
The card will be available for *n* days
(until *expiration date*).

The message for the sender looks like this:

To: *sender*
From: *sender*
Subject: Your card for *recipient*
This message is for your records. You sent an electronic greeting card to:
recipient

> You can view the card with your Web browser at the following address:
> *URL*
> The card will be available for *n* days
> (until *expiration date*).

The sender and recipient names and email addresses are contained in the card hash, and `$desc_date` indicates the expiration date in descriptive form. But what address should we use in the `From:` header, and what should the URL for requesting the card look like?

The `From:` address should be some valid address in case the person receiving the message attempts to reply to it. One choice is to have the message appear to come from the card sender, which is what `make_ecard.pl` does. (An alternative would be to have messages appear to come from your card-sending service, using an address such as `e-cards@snake.net`. If you use a special address, you'll have to set up an account to receive mail sent to it, or alias it to someone else.)

The more difficult thing is figuring out what kind of URL to include in the messages so that the card sender and recipient can view the card. We'll use another script, `show_ecard.pl`, for retrieving and displaying the card, so the initial part of the URL has to name that script. If we assume that `make_ecard.pl` and `show_ecard.pl` are both installed in the same directory on your Web server, `make_ecard.pl` can get its own URL and use it to figure out the URL for `show_ecard.pl` by replacing the last component (everything after the last slash) with "show_ecard.pl":

```
$url = url ();
$url =~ s/[^\/]+$/show_ecard.pl/;
```

We'll also need to add some information to the URL that identifies the card. The ID number does this, but we really need something more than just the ID by itself. Remember that we want to be able to tell when the recipient requests the card for the first time so that we can mark the card record in the database as having been viewed. We also may need to notify the sender when that happens. That means we need to know who is requesting the card. Also, we want to discourage casual card browsing by people other than the sender or recipient. (That is, we don't want other people sending requests for arbitrary card ID numbers, to see what cards people are sending. We're not guaranteeing that cards are private, but the intent isn't really to provide each card as a completely public resource, either.)

One simple way to tell whether the sender or the recipient is requesting a card and to discourage browsing by outsiders is to use card-retrieval URLs that identify the role and address of the requester. If card 407 was sent by the U.S. President to the First Lady, the URLs that would be provided to each of them for viewing the card would look like this:

```
.../show_ecard.pl?id=407;sender=president@whitehouse.gov
.../show_ecard.pl?id=407;recip=first.lady@whitehouse.gov
```

These URLs are constructed from the base URL by appending the appropriate parameter values:

```
$recip_url = $url . sprintf ("?id=%d;recip=%s",
                    $card_ref->{id}, escape ($card_ref->{recip_email}));
$sender_url = $url . sprintf ("?id=%d;sender=%s",
                    $card_ref->{id}, escape ($card_ref->{sender_email}));
```

With the URLs in hand, we have all the information we need to generate the email messages and send them. `make_ecard.pl` does so using the `sendmail()` function from the `Mail::Sendmail` module discussed earlier in the chapter.

Retrieving Cards for Display

Requests to see cards are handled by the `show_ecard.pl` script. For purposes of reading a card from the database, the script needs only the card ID number:

```
SELECT * FROM ecard WHERE id = n
```

Given that we want only the card sender and recipient to view the card, however, we'll also require that the email address in the URL match the appropriate email address in the card record. If the recipient requests the card, for example, the query looks like this:

```
SELECT * FROM ecard WHERE id = n AND recip_email = 'address'
```

For the sender, the query checks the `sender_email` column rather than the `recip_email` column. Given this mechanism, `show_ecard.pl` acts as follows:

- Look in the URL for the `id` parameter that identifies the card, and either a `sender` or `recip` parameter that identifies the role and email address of the person who wants to see it.
- Look up the record that matches the `id` value and the email address from either the `sender_email` or `recip_email` column (depending on which one was specified in the URL). This way casual attempts to view cards will fail. (To hack in, you would have to know not only a card's ID number, but also who sent it or to whom it was sent.)
- Display the card. `show_ecard.pl` generates a page containing the text of the card and, if there is a picture, an `<img>` tag that is handled by `serve_image.pl`.
- If the card's `viewed` column is `NULL` and the requester is the recipient, this is the first time the recipient has asked to see the card. Set the `viewed` value to the current date and, if the sender has requested notification, send an email message confirming that the recipient has taken a look at the card.

The first part of `show_ecard.pl` checks the URL parameters, determines whether they're valid, and displays the card if so:

```
print header (),
    start_html (-title => "View Your Electronic Greeting Card",
                -bgcolor => "white");

my $id = param ("id");
my $sender = param ("sender");
my $recip = param ("recip");
my $valid = 0;

if (defined ($id))          # got the card ID, look for sender or recipient
{
    if (defined ($sender))
    {
        $valid = 1;
        show_ecard ($id, $sender, "sender_email");
    }
    elsif (defined ($recip))
    {
        $valid = 1;
        show_ecard ($id, $recip, "recip_email");
    }
}
if (!$valid)
{
    print p ("Missing or invalid e-card parameters specified.\n"
            . "Please check the URL that was sent to you.");
}

print end_html ();
```

Assuming the card ID value and an email address are present in the URL, `show_ecard()` looks up the card from the database and displays it:

```
sub show_ecard
{
my ($id, $email, $col_name) = @_;
my $dbh = WebDB::connect ();
my ($sth, $ref);

    $sth = $dbh->prepare (
                "SELECT * FROM ecard
                WHERE id = ? AND $col_name = ?");
    $sth->execute ($id, $email);
    $ref = $sth->fetchrow_hashref ();
    $sth->finish ();
    if (!$ref)
    {
        $dbh->disconnect ();
        print p ("Sorry, card was not found; perhaps it has expired.");
        return;
    }
```

continues

continued

```
    # Print recipient name and email.  If a picture was selected,
    # generate an <img> tag for it.  Then display the message text and
    # sender information.

    print p (escapeHTML ("To: $ref->{recip_name} ($ref->{recip_email})"));
    if ($ref->{picture} ne "")
    {
        print img ({-src => "serve_image.pl?name=" . escape ($ref->{picture}),
                    -alt => escapeHTML ($ref->{picture})});
    }
    print p (escapeHTML ($ref->{message}));
    print p (escapeHTML ("This message was sent to you by: "
                    . "$ref->{sender_name} ($ref->{sender_email})"));

    # If this is a request by the recipient, set the "viewed" date if
    # it hasn't yet been set; notify the sender that the recipient has
    # viewed the card if the sender requested notification. Also,
    # display some links for replying to to sender by email or for
    # generating a reply card.

    if ($col_name eq "recip_email")
    {
        my $mail_url = sprintf ("mailto:%s?subject=%s",
                                escape ($ref->{sender_email}),
                                escape ("Thanks for the e-card"));
        print hr ();
        print a ({ -href => $mail_url }, "Send mail to sender") , br ();
        print a ({ -href => "make_ecard.pl" }, "Create your own e-card");

        if (!$ref->{viewed})
        {
            $dbh->do ("UPDATE ecard SET viewed=CURRENT_DATE WHERE id = ?",
                    undef, $ref->{id});
            notify_sender ($ref) if $ref->{sender_notify} eq "Y";
        }
    }
    $dbh->disconnect ();
}
```

The `notify_sender()` function generates email to let the sender know the recipient has looked at the card:

```
sub notify_sender
{
my $ref = shift;        # card record
my %mail;
```

```
    $mail{To} = $mail{From} = $ref->{sender_email};
    $mail{Subject} = "Your e-card for $ref->{recip_name}";
    $mail{Message} = "
Your card to $ref->{recip_name} ($ref->{recip_email})
has been viewed by the recipient.";
    sendmail (%mail);
}
```

Expiring Old Cards

The `expiration` column is present in the `ecard` table to allow old cards to be removed; there's no need to keep them around forever. Cards won't delete themselves, however, so we need to set up a mechanism to handle that task. This can be done by setting up a `cron` job to identify and delete cards whose expiration date has passed. If I want to expire cards at 1:15 a.m. each morning using a script `expire_ecard.pl` installed in my `bin` directory, for example, I'd add a line like this to my `crontab` file:

```
15 1 * * * /u/paul/bin/expire_ecard.pl
```

The expiration script itself can be written different ways. The following version deletes the expired records, but also prints a message indicating how many records were deleted. (As usual, I'm assuming that `cron` will mail to me the output of any programs it runs on my behalf.)

```
#! /usr/bin/perl -w
# expire_ecard.pl - remove greeting cards that have expired

use strict;
use lib qw(/usr/local/apache/lib/perl);
use WebDB;

my $dbh = WebDB::connect ();    # connect to database
my $count = $dbh->do ("DELETE FROM ecard WHERE expiration < CURRENT_DATE");
$count += 0;    # convert string to number, in case it's "0E0"
print "$count e-cards have expired and were deleted\n";
$dbh->disconnect ();
exit (0);
```

Suggested Modifications

Our card-sending application is the most complex of the chapter. Nevertheless, it's relatively unsophisticated, compared to some of the big sites devoted to electronic greeting cards, and there are a lot of things you could add to it. A couple of obvious additions would be to allow delivery to multiple recipients, or to allow the delivery date to be set. As written, the card is sent to a single recipient, and it's sent as soon as the user selects the Send Card button. You could also implement card categories such as "get well," "sympathy," "good luck", "birthday," or "wedding." If you had such categories, you could add a SET column to the image table that would indicate which card categories each image applies to, enabling you to present category-specific image galleries. Other modifications could focus on enhancing the existing features in various ways. If you have lots of images, for example, you'd probably want to present the picture gallery using multiple pages instead of displaying all pictures on the same page.

When a visitor first begins to create a card, make_ecard.pl generates a new ID number so the card can be tracked through each part of the process. At the end of the process, we assign an expiration date. If the user never completes the card, however, the expiration date remains NULL. That's a problem: The expiration mechanism is based on the value of the expiration date, so uncompleted cards never get expired. How would you address this problem, making sure not to remove records for cards that visitors currently are working on?

After a visitor completes the card construction process, it's not possible for someone else to come along and modify the card. (make_ecard.pl will notice that the expiration date has been set, which indicates that the card is finished.) However, card hijacking is possible while the card is being built, between the time that the card ID generated and the time the expiration date is assigned. How might you deal with this?

Modify expire_ecard.pl to provide information that indicates how many of the expired cards actually were viewed by the recipient.

In this chapter, you've seen how to build several interactive applications that run from your browser. For some of these applications, we attempted to reduce the amount of work involved in generating and processing the form by storing information about it in a data structure. In the product-registration script at the beginning of the chapter, for example, we used an array to list the names, labels, and sizes of the text-input fields, as well as whether each field had to have a non-empty value at form-submission time. In Chapter 6, we'll further explore the potential for deriving information about forms in a way that can be used automatically. For applications that are tied to tables in your database, one good source of information that can be used in relation to form processing is the knowledge that MySQL itself has about the structure of the tables in your database. As we'll see, this information can be used in several ways to make your Web programming tasks easier.