

9

Building Databases

In previous chapters, you created a very nice movie review Web site, but now the hand-holding is over, my friend. It's time for us to push you out of the nest. In this chapter, you will have the opportunity to create your own databases, and your own Web site.

We show you how to put together a comic book appreciation Web site, but you can certainly take the concepts we teach you and branch off to create that online auction or antique car site you have always dreamed about. We think the comic book appreciation Web site is cooler, but *whatever*. You do your thing.

In this chapter, we are going to cover the basics of creating your own database. Topics we discuss include:

- Planning the design of your database
- Database normalization
- Creating your database
- Creating and modifying tables in your database
- Building Web pages to access your data with PHP

Getting Started

You have a great idea for a site, right? Excellent. Open up your PHP editor and start coding! Believe it or not, many people approach the creation of a Web site in just this way. You may be tempted to yourself. It is not impossible to create a good site in this manner, but you are seriously handicapping your chances for greatness. Before you begin, you need a plan.

We're not going to tell you how to plan out an entire Web site, complete with charts and maps and business models. That's not what this book is all about. We are going to assume that you or somebody in your company has already done that by reading other great books on business models, attending seminars, reading great articles on the Web, and perhaps even hiring a business

Chapter 9

consultant who will help you with everything but building the site (because that's what we're going to teach you how to do).

So you have a great idea for a Web site *and* a plan. What do you suppose is the first step in creating a successful Web application using PHP, Apache, and MySQL? We'll give you a clue: Look at the title of this chapter.

We need to build the database this site will be based on. Don't worry—one of the great things about relational database design is that you don't have to create *every* table your site will use. You can start with a few, and build on it. As long as you follow the basic principles of good database design, your database should be quite scalable (that is, expandable to any size).

Nam et Ipsa Scientia Potestas Est!

That is, *knowledge is power*. Very profound words, coming from a man who wore a big, ruffled collar. Francis Bacon coined the phrase 400 years ago, and it still holds true today.

Of course, information is the foundation of knowledge. Knowledge consists of having information available to you and knowing what to do with it. Data is the building blocks—the facts and figures—that we piece together to create useful sets of information.

We must be sure to store this data in an easily accessible place and in a way that allows us to relate that data to any other data fairly easily. We also want to be able to modify or remove each piece of data quickly and efficiently, without disturbing other data. With proper database design, all of this is possible.

Sound like a daunting task? Don't worry. You see, we know a secret that has been kept hidden like the magician's code: *Efficient database design is easy*. No, really, we promise! You see, most of us computer geeks like to seem invaluable and very intelligent, and it sounds quite impressive to most interviewers to see on a resume "Designed a comprehensive Web site utilizing an RDBMS backend." When you are done with this chapter, you will be able to put that on your resume as well!

What Is a Relational Database?

Let's first cover a few basics of database design. The relational database is a concept first conceived by E. F. Codd of IBM, in 1970. It is a collection of data organized in tables that can be used to create, retrieve, delete, and update that data in many different ways. This can be done without having to reorganize the tables themselves, especially if the data is organized efficiently.

Take a look at the first table that follows. You can see that we have a very simple collection of data consisting of superheroes' aliases and real names, and their superhero ID. Nothing too amazing, of course, but notice how we relate it to the league table that follows it. Each superhero user has a `League_ID` that corresponds to an ID in the league table. Through this link, or *relationship*, you can see that Average Man is a member of the Dynamic Dudes League because the ID in the league table matches his `League_ID` in the superhero table.

Building Databases

ID	League_ID	Alias	Real Name
1	2	Average Man	Bill Smith
2	2	The Flea	Tom Jacobs
3	1	Albino Dude	George White
4	3	Amazing Woman	Mary Jones

ID	League
1	Extraordinary People
2	Dynamic Dudes
3	Stupendous Seven
4	Justice Network

At first glance, it may seem silly to create a table with one data column and an ID. Why not just put the league name in the superhero table? Imagine that you had a database of 10,000 superheroes, and 250 of them were in the Dynamic Dudes league. Now imagine that the Superhero Consortium decided to do a reorganization and “Dynamic Dudes” was changed to the “Incredible Team.” If the league name were in the superhero table, you would have to edit 250 records to change the value. With the leagues in a separate, *related* table, you have to change the name in only one place.

That is the key to a relational database. And speaking of keys . . .

Keys

A key is a column that identifies a row of data. In the superhero table, the first column is a key called “ID,” as it is in the league table. In both cases, because they are unique, and in the table of the data they represent, they are called *primary keys*.

Most of the time, the primary key is a single column, but it is not uncommon to use more than one column to make up a primary key. The important distinction is that for each row, the primary key must be unique. Because of that characteristic, we can use the key to identify a specific row of data.

The primary key must contain the following characteristics:

- They cannot be empty (null).
- They will never change in value. Therefore, a primary key cannot contain information that might change, such as part of a last name (for example, smith807).
- They must be unique. In other words, no two rows can contain the same primary key.

Chapter 9

The `League_ID` column in the `superhero` table is also a key. It matches the primary key of the `league` table, but it is in a different, or *foreign*, table. For this reason, it is called a foreign key. Although not a requirement, many programmers will give the foreign key a name that identifies what table it refers to (“League”), and some identifier that marks it as a key (“_ID”). This, along with the fact that keys are usually numeric, makes it fairly clear which column is the foreign key, if one exists in the table at all.

Keys do not have to be purely numeric. Other common values used as primary keys include Social Security numbers (which contain dashes), e-mail addresses, and ZIP Codes. Any value is valid as a primary key as long as it is guaranteed to be unique for each individual record in the table, and will not change over time.

Keys can enable your tables to be recursive. You might, for example, have a `sidekick_ID` column in the `superhero` table that we could link to the `ID` column in the same table. Sidekicks are heroes, too, you know . . .

Relationships

In order to be related, the two tables need a column they can use to tie them together. The `superhero` and `league` tables are related to each other by the `League_ID` column in the `superhero` table, and the `ID` field in the `league` table. There is no explicit link created in the database; rather, you create the relationship by linking them with a SQL statement:

```
SELECT * FROM superhero s, league l WHERE s.League_ID = l.ID
```

In plain English, this statement tells the MySQL server to “select all records from the `superhero` table (call it ‘s’) and the `league` table (call it ‘l’), and link the two tables by the `superhero League_ID` column and the `league ID` column.”

There are three types of relationships: one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N). Our previous example is a one-to-many relationship. To figure out what type of relationship the tables have, ask yourself how many superheroes you can have in a league. The answer is more than one, or “many.” How many leagues can a superhero belong to? The answer is “one.” That is a one-to-many relationship. (Of course, in some universes, a superhero might belong to more than one league. But for our example, our superheroes exhibit league loyalty.)

One-to-many is the most common database relationship. Such 1:1 relationships don’t happen often, and a many-to-many relationship is actually two one-to-many relationships joined together with a linking table. We explore that further later in the chapter.

Although they are more rare, here’s an example of a one-to-one (1:1) relationship just so you know. Say you have a link between a company and its main office address. Only one company can have that exact address. In many applications, however, the main office address is included in the company table, so no relationship is needed. That’s one of the great things about relational database design. If it works for your needs, then there is no “wrong” way to do it.

Referential Integrity

The concept of referential integrity may be a little lofty for a beginner book like this, but we think it is important to touch on this briefly. If your application has referential integrity, then when a record in a table refers to a record in another table (as the previous example did), the latter table will contain the corresponding record. If the record is missing, you have lost referential integrity.

In many cases, this is not disastrous. You might have an article written by an author whose name no longer exists in the author table. You still want to keep the article, so losing the referential integrity between authors and articles is okay. However, if you have an order in your database that can't be related to a customer because the customer was deleted, then you might be hard pressed to figure out where to send the product, and who to charge for it!

There are ways to enforce referential integrity in a MySQL database. However, these concepts and procedures are beyond the scope of this book. If you are interested in obtaining more information about referential integrity and foreign keys, visit www.mysql.com/doc/en/InnoDB_foreign_key_constraints.html.

Normalization

“Database normalization” is one of those big fancy terms that database administrators like to throw around, along with “Boyce-Codd Normal Form,” “trivial functional dependency,” and “Heisenberg compensator.” They aren't really important terms to know to be able to design a good database, but we'll touch on normalization here.

For our purposes, we will simply define normalization as the process of modifying your database table structure so that dependencies make sense, and there is no redundant data. In a moment, we are going to go through this process. The best way to learn is to do!

Designing Your Database

It's time to design your application. This will be a relatively simple application, but it will help you learn important concepts such as normalization and expose you to various SQL commands.

Typically, this is where we would take you through a “Try It Out” section and tell you How It Works. When first designing a database, however, you do not need your computer. All you need is a pad of paper and a pencil. So, go get a pad of paper and a pencil. We'll wait.

Let's draw some tables.

The application you are going to design is a comic book character database. You will store a little bit of information about various characters, such as their alter ego's alias, their real names, the powers they possess, and the location of their lair. (Yes, that's right. I said “lair.”)

Chapter 9

Creating the First Table

Before we open MySQL and start mucking around with tables, we need to figure out how we are going to store all of the data. For simplicity, let's create one big table with all of the relevant data. You can draw it out on your piece of paper, or if you just can't stay away from your computer, use your favorite spreadsheet program. Copy the information you see in the table that follows.

name	real name	power 1	power 2	power 3	lair address	city	st	zip
Clean Freak	John Smith	Strength	X-ray vision	Flight	123 Poplar Avenue	Townsbury	OH	45293
Soap Stud	Efram Jones	Speed			123 Poplar Avenue	Townsbury	OH	45293
The Dustmite	Dustin Huff	Strength	Dirtiness	Laser vision	452 Elm Street #3D	Burgtown	OH	45201

We'll call that table "zero," because we're not even at the first step yet, and that data is just *ugly* (from a relational database standpoint).

The first thing you should notice is that there are multiple power columns. What would you do if you had to add a character with more than three powers? You would have to create a new column, and that's not good. Instead, let's combine all the powers into one column, and then separate each power into its own separate row. The other columns are duplicated in these additional rows (so, Clean Freak would have three rows instead of one, each row including a different power in the power column, but the name, address, and so on would remain identical among the three listings). This concept is called *atomicity*. Each value (cell) is *atomic*, or has only one item of data.

Let's also create a unique primary key for each character. Yes, you could use the character's name, but remember that a primary key should never be something that could change, and it must be unique. To handle this requirement we'll create an ID column.

Because in this pass we have multiple rows with the same character and the multiple rows are a result of the existence of multiple powers, we'll combine the ID column with the power column to create the primary key. When more than one column makes up the primary key, it is called a *composite primary key*. We'll mark the primary key columns with an asterisk (*) to highlight them for you.

Your table should look like the one that follows. We'll call this table "one" because it's our first pass at normalizing. (Yes, you are in the middle of a normalization process. We told you it wasn't difficult.)

id*	name	real name	power*	lair address	city	st	zip
1	Clean Freak	John Smith	Strength	123 Poplar Avenue	Townsbury	OH	45293
1	Clean Freak	John Smith	X-ray vision	123 Poplar Avenue	Townsbury	OH	45293
1	Clean Freak	John Smith	Flight	123 Poplar Avenue	Townsbury	OH	45293
2	Soap Stud	Efram Jones	Speed	123 Poplar Avenue	Townsbury	OH	45293

Building Databases

id*	name	real name	power*	lair address	city	st	zip
3	The Dustmite	Dustin Hare	Strength	452 Elm Street #3D	Burgtown	OH	45201
3	The Dustmite	Dustin Hare	Dirtiness	452 Elm Street #3D	Burgtown	OH	45201
3	The Dustmite	Dustin Hare	Laser vision	452 Elm Street #3D	Burgtown	OH	45201

Looking better, but there is still repeated data in there. In fact, the power column is what is causing the duplicate data. Let's separate out the power column and use a foreign key to relate it to the original table. We will also further normalize the power table so that we get rid of duplicate data. This is pass number "two." See the three tables that follow.

id*	name	real name	lair address	city	st	zip
1	Clean Freak	John Smith	123 Poplar Avenue	Townsburg	OH	45293
2	Soap Stud	Efram Jones	123 Poplar Avenue	Townsburg	OH	45293
3	The Dustmite	Dustin Hare	452 Elm Street #3D	Burgtown	OH	45201

id*	power
1	Strength
2	X-ray vision
3	Flight
4	Speed
5	Dirtiness
6	Laser vision

char_id*	power_id*
1	1
1	2
1	3
2	4
3	1
3	5
3	6

As you can see, we have much less repeated data than we did before. The powers have been separated out, and a link table has been created to link each power to each appropriate character.

Chapter 9

It may seem a bit nitpicky, but you still have some duplicate data that you can take care of in the character table. It is quite possible for more than one character to be in the same lair, as is the case with Clean Freak and Soap Stud. Let's create a lair table, and link it to the character table with keys. Let's also add a new column to the character table for alignment. See the two tables that follow.

id*	lair_id	name	real name	align
1	1	Clean Freak	John Smith	Good
2	1	Soap Stud	Efram Jones	Good
3	2	The Dustmite	Dustin Hare	Evil

id*	lair address	city	st	zip
1	123 Poplar Avenue	Townsburg	OH	45293
2	452 Elm Street #3D	Burgtown	OH	45201

We waited to add the alignment column to illustrate a point. If you are in the middle of the normalization process, and discover that there is some other data you need to add, it isn't difficult to do so. You could even add a completely new table if you needed to. That is the great thing about relational database design.

The City and State fields are not only duplicates, but they are redundant data with the ZIP Code (which is in itself a representation of the City/State). City and State are also not directly related to the lairs (because other lairs could exist in the same city). For these reasons, we will put City and State in a separate table. Because the ZIP Code is numeric, and a direct representation of City/State, we will make the Zip column a primary key. This is pass "three," shown in the three tables that follow.

id*	lair_id	name	real name	align
1	1	Clean Freak	John Smith	Good
2	1	Soap Stud	Efram Jones	Good
3	2	The Dustmite	Dustin Hare	Evil

id*	zip_id	lair address
1	45293	123 Poplar Avenue
2	45201	452 Elm Street #3D

id*	city	st
45293	Townsburg	OH
45201	Burgtown	OH

Building Databases

You may have noticed that we have created a many-to-many (M:N) relationship between the characters and their powers (a character can have multiple powers, and many characters may have the same power). There are two tables with primary keys, and a linking table between them has two foreign keys, one for each of the tables. The combination of the foreign keys is a primary key for the `char_power` table. This enables the M:N relationship.

Just for fun, let's add a small table that links the superheroes to villains, and vice versa. This is another M:N relationship because any superhero can have multiple villain enemies, and any villain can have multiple superhero enemies. Of course, we have the character table as one of the "many" sides of the equation—can you figure out what table we will use for the other "many" side? If you said the character table, you are correct! This is just like the character-power relationship, but this time we reference the table to itself via a `good_bad` linking table. The `goodguy_id` and `badguy_id` columns *each* link to the `id` column in the character table. Each column in the `good_bad` table is a foreign key, and both columns make up a composite primary key.

<code>goodguy_id*</code>	<code>badguy_id*</code>
1	3
2	3

And just like that, you have created your database design. Congratulations! You now have a "map" that will help you create your database tables on the server. Not only that, but you just normalized your database design as well by modifying your database table structure so that dependencies make sense, and there is no redundant data. In fact, you have actually gone through the proper normalization steps of First, Second, and Third Normal Form.

What's So Normal About These Forms?

Remember we told you that we were calling the first table "zero"? That's called Zero Form. It is basically the raw data, and is usually a very flat structure, with lots of repeated data. You see data like this sometimes when a small company keeps records of its customers in a spreadsheet.

The first pass through the table, which we called pass "one," was the first step of normalization, called "First Normal Form," or 1NF. This step requires that you eliminate all repeating data in columns (which we did with the power column), create separate rows for each group of related data, and identify each record with a primary key. Our first step satisfies the requirements of 1NF.

You can see where we're going with this, can't you? The Second Normal Form (2NF) requirements state that you must place subsets of data in multiple rows in separate tables. We did that by separating the power data into its own table. Second Normal Form also requires that we create a relationship with the original table by creating a foreign key. We did that in pass "two," when we satisfied the requirements for 2NF.

On our third pass, we removed all the columns not directly related to the primary key (City and State), and used the ZIP Code as the foreign key to the new `city_state` table. Third Normal Form (3NF) is then satisfied. Congratulations! You normalized a database just like the pros do.

Chapter 9

There are further requirements for database normalization, but Third Normal Form is generally accepted as being good enough for most business applications. The next step is Boyce-Codd Normal Form, followed by Fourth Normal Form and Fifth Normal Form. In our case, the other Forms don't apply—the database is as normalized as it needs to get. All tables are easily modifiable and updateable, without affecting data in the other tables.

We know there are some database gurus out there who would tell you that in order to completely satisfy the Forms of normalization, that the align column should be put into its own table and linked with a foreign key. While that may be true in the strictest sense of the rules, we usually think of normalization as a guideline. In this case, we have only two values, good and evil. Those values will never change, and they will be the only values available to the user. Because of this, we can actually create a column with the ENUM datatype. Because the values good and evil will be hardcoded into the table definition, and we don't ever see a need to change the values in the future, there is no problem with keeping those values in the char_main table.

Standardization

When you are designing a new application, it is a very good idea to come up with standards, or design rules, that you adhere to in all cases. These can be extensive, such as the standards published by the W3C for HTML, XML, and other languages. They can be very short, but very strict, such as the list of ten standards brought down from a mountain by an old bearded man. For now we'll just standardize our table structure. For this application, we came up with the following table standards:

- ❑ **Table names:** Table names should be descriptive, but relatively short. Table names will be in lowercase. They should describe what main function they serve, and what application they belong to. All six of our tables should start with "char_" to show that they belong to the character application.
- ❑ **Column names:** Table columns are similar to table names. All column names will be in lowercase. They will be kept short, but multiple words (such as lair and address) will be separated by an underscore "_" (lair_addr).
- ❑ **Primary keys:** Single primary keys will always be called "id". Except in special cases, primary keys will be an integer datatype that is automatically incremented. If they consist of a single column, they will always be the first column of the table.
- ❑ **Foreign keys:** Foreign keys will end with "_id". They will start with the table descriptor. For example, in the char_lair table, the foreign key for the char_zipcode table will be called zip_id.

Finalizing the Database Design

One other thing we like to do during the database design process is put the datatypes into the empty cells of each table. We can print these tables and easily refer to them when we are writing the SQL code. You may want to do this yourself (or just use the tables provided).

If you don't understand datatypes, you can learn about them in Chapter 3, and we discuss datatypes in more detail a little later in this chapter as well. For now, just understand that datatypes are the type of data stored in each table column, such as INT (integer), VARCHAR (variable-length character string), or ENUM (enumerated list). When appropriate, they are followed by the length in parentheses; for example, varchar(100) is a character column that can contain up to 100 characters.

Building Databases

If you have been working in a spreadsheet, simply erase all of the actual data in those tables. If you used a pad and pencil, just follow along. Reduce the tables to two rows, one with column names, the other row blank. If you want, you can make a copy before erasing the data.

In keeping with the previously listed table standards, we arrive at the following tables. Yours should look very similar.

id*	lair_id	name	real_name	align
int(11)	int(11)	varchar(40)	varchar(80)	enum('good', 'evil')

id*	power
int(11)	varchar(40)

char_id*	power_id*
int(11)	int(11)

id*	zip_id	lair_addr
int(11)	varchar(10)	varchar(40)

id*	city	state
varchar(10)	varchar(40)	char(2)

good_id*	bad_id*
int(11)	int(11)

We think it is about time we actually created these tables on the server. Ready? Not so fast: We have to create the database first.

Creating a Database in MySQL

There are a number of ways to create a database. All require the execution of a SQL statement in one way or another, so let's look at that first:

```
CREATE DATABASE yourdatabase;
```

Chapter 9

What, were you expecting something more complicated? Well, an optional parameter is missing: `IF NOT EXISTS`. We're pretty sure you know whether or not it exists, but if it makes you feel better, you can certainly add that:

```
CREATE DATABASE IF NOT EXISTS yourdatabase;
```

That's all there is to it. Think of the database as an empty shell. There is nothing special about it, really. The interesting stuff comes later, when you create the tables and manipulate the data.

That said, we still have to figure out how we are going to execute that SQL statement. Here are a few suggestions:

- ❑ From the MySQL command prompt. Do it this way only if you have access to the server on which MySQL is installed. If you are running your own server, or you have telnet access to the server, this may be an option for you.
- ❑ If you are being hosted by an ISP, you may need to request that the ISP create a database for you. For example, on one author's site the ISP has CPanel installed, and he simply clicks the module called MySQL Databases. From the next page, he simply types in the database he wants to create and clicks a button, and it's created for him.

ISPs will usually give you this option because you have a limit in your contract on how many databases you are allowed to create. On one of our sites, for example, the limit is ten databases.

- ❑ If you have PHPMyAdmin installed (either on your own server or through your ISP), you can then run the SQL command from there. PHPMyAdmin is a valuable tool, and we recommend you use it if that is an option for you. It allows you to see your table structures and even browse data. It is a dangerous tool, however, because you can easily drop tables or entire databases with the click of a button, so use it carefully.
- ❑ Another option is to run your SQL statement from a PHP file. Most likely, if you are hosted by an ISP, they won't allow the creation of databases in this manner. However, almost any other SQL statement will work using this method. This is the way we will be running SQL commands through the rest of this chapter.

Once you have determined how you are going to run that SQL command, go ahead and do it. Make sure you substitute your own database name for `yourdatabase`. Because we are going to develop a comic book appreciation Web site, you could call it `comic_book_app`:

```
CREATE DATABASE IF NOT EXISTS comic_book_app;
```

Now that we have a design mapped out and a database created in MySQL, it is time to create some tables.

Try It Out Create the Table

First, we're going to create the file that will hold the hostname, username, password, and database values.

1. Open your favorite text editor, and enter the following code (making sure you use the proper values for your server):

```
<?php

define('SQL_HOST','yourhost');
define('SQL_USER','joeuser');
define('SQL_PASS','yourpass');
define('SQL_DB','yourdatabase');

?>
```

2. Save the file as `config.php`.

This file will be included on each subsequent PHP file that needs to access the database.

3. Type the following code in your favorite PHP editor, and save it as `make_table.php`:

```
<?php
require('config.php');

$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());

mysql_select_db(SQL_DB, $conn);

$sql1 =
"CREATE TABLE IF NOT EXISTS char_main (
    id int(11) NOT NULL auto_increment,
    alias varchar(40) NOT NULL default '',
    real_name varchar(80) NOT NULL default '',
    lair_id int(11) NOT NULL default 0,
    align enum('good','evil') NOT NULL default 'good',
    PRIMARY KEY (id)
)";

$sql2 =
"CREATE TABLE IF NOT EXISTS char_power (
    id int(11) NOT NULL auto_increment,
    power varchar(40) NOT NULL default '',
    PRIMARY KEY (id)
)";

$sql3 =
"CREATE TABLE IF NOT EXISTS char_power_link (
    char_id int(11) NOT NULL default 0,
    power_id int(11) NOT NULL default 0,
    PRIMARY KEY (char_id, power_id)
)";

$sql4 =
"CREATE TABLE IF NOT EXISTS char_lair (
    id int(11) NOT NULL auto_increment,
```

Chapter 9

```

zip_id varchar(10) NOT NULL default '00000',
lair_addr varchar(40) NOT NULL default '',
PRIMARY KEY (id)
)";

$sql5 =
"CREATE TABLE IF NOT EXISTS char_zipcode (
  id varchar(10) NOT NULL default '',
  city varchar(40) NOT NULL default '',
  state char(2) NOT NULL default '',
  PRIMARY KEY (id)
)";

$sql6 =
"CREATE TABLE IF NOT EXISTS char_good_bad_link (
  good_id int(11) NOT NULL default 0,
  bad_id int(11) NOT NULL default 0,
  PRIMARY KEY (good_id,bad_id)
)";

mysql_query($sql1) or die(mysql_error());
mysql_query($sql2) or die(mysql_error());
mysql_query($sql3) or die(mysql_error());
mysql_query($sql4) or die(mysql_error());
mysql_query($sql5) or die(mysql_error());
mysql_query($sql6) or die(mysql_error());
echo "Done.";
?>

```

4. Run this file by loading it in your browser.

Assuming all goes well, you should see the message “Done” in your browser. The database now should contain all six tables.

How It Works

Every PHP script that needs to access your database on the MySQL server will include `config.php`.

These constants will be used in your scripts to gain access to your database. By putting them here, in one file, you can change the values any time you move servers, change the name of the database, or change your username/password. Any time you have information or code that will be used in more than one PHP script, you should include it in a separate file. That way, you’ll need to make your changes in only one location.

```

define('SQL_HOST', 'yourhost');
define('SQL_USER', 'joeuser');
define('SQL_PASS', 'yourpass');
define('SQL_DB', 'yourdatabase');

```

The `make_tables.php` file is a one-time script: You should never have to run it again, unless you needed to drop all of your tables and recreate them. So, rather than explain all of the code in the page, let’s just look at one of the SQL statements:

```
CREATE TABLE IF NOT EXISTS char_main (
  id int(11) NOT NULL auto_increment,
  alias varchar(40) NOT NULL default '',
  real_name varchar(80) NOT NULL default '',
  lair_id int(11) NOT NULL default 0,
  align enum('good','evil') NOT NULL default 'good',
  PRIMARY KEY (id)
)
```

The syntax for creating a table in SQL is the following:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  [(create_definition,...)] [table_options] [select_statement]
```

Obviously, we are not using the `TEMPORARY` keyword. We want this table to be permanent and exist after our connection with the database. We are using the `IF NOT EXISTS` keyword, but only if this page is loaded twice. If you attempt to load the page again, MySQL will not attempt to re-create the tables, and will not generate an error.

Our table name in this case is `char_main`. The columns we create are `id`, `alias`, `real_name`, `lair_id`, and `align`, which are the names we came up with earlier.

Let's look at each column:

- ❑ `id int(11) NOT NULL auto_increment`: The `id` column is set as an integer, with 11 maximum places. An integer datatype can contain the values -2147483648 to 2147483648. A sharp observer would note that the max value is only ten digits. The eleventh digit is for negative values.

`NOT NULL` will force a value into the column. With some exceptions, numeric columns will default to 0, and string columns will default to an empty string (' '). Very rarely will we allow a column to carry a `NULL` value.

 The code `auto_increment` causes the column to increase the highest value in the table by 1, and store it in this column. A column set to `auto_increment` does not have a default value.
- ❑ `alias varchar(40) NOT NULL default ''`: the `alias` column is set as a `varchar` datatype, a string of 0 to 255 characters. We are allotting 40 characters, which should be enough for any character name. A `varchar` differs from a `char` datatype by the way space is allotted for the column.

 A `varchar` datatype occupies only the space it needs, whereas `char` datatypes will always take up the space allotted to them by adding spaces at the end. The only time you really need to use the `char` datatype is for strings of less than three characters (such as the `State` column in the `char_zipcode` table).
- ❑ `real_name varchar(80) NOT NULL default ''`: Similar to `alias`. We are allotting 80 characters, which should be enough for our needs.
- ❑ Note that we did not separate the `real_name` column into `first_name` and `last_name` columns. If you wanted to do that, you certainly could, but in this small application it really isn't necessary. On the other hand, in a human resources application for your company, having separate columns for first and last name is almost a requirement, so that you can do things such as greet employees by their first names in a company memo.

Chapter 9

- ❑ `lair_id int(11) NOT NULL default 0`: Our foreign key to the `char_lair` table is also an integer of length 11, with a default value of 0.
- ❑ `align enum('good','evil') NOT NULL default 'good'`: the `align` column can be one of two values: “good” or “evil.” Because of this, we use an `enum` datatype, and default it to “good.” (Everyone has some good in them, right?)

You now have a database. You have tables. If you just had a way to enter some data into your tables in your database, you’d have an application you could give to your users, where they could store information about their favorite superheroes and villains.

You could enter the data with some query statements in PHPMyAdmin, but that probably wouldn’t be too efficient, not to mention that your users wouldn’t have any access to it. You need some sort of interface for them that they can use to create and edit data.

Let’s design some Web pages for them.

Creating the Comic Character Application

It’s back to the drawing board. Literally. Get away from your computer. We’re going to put together some ideas for a Web application.

First of all, you need a page to display a list of comic book characters along with some information about them. It doesn’t need to include every detail about them (such as the location of their secret lair), but it should have enough data so that users can distinguish who they are and read a little bit of information about them.

We will list the following information:

- ❑ Character name (alias)
- ❑ Real name
- ❑ Alignment (good or evil)
- ❑ Powers
- ❑ Enemies

We also need a character input form. This form will serve two purposes. It will allow us to create a new character, in which case the form will load with blank fields and a `create` button, or it will allow us to edit an existing character, in which case it will load with the fields filled in, and an `update` button. We’ll also have a `reset` button that either clears the new form, or restores the edited form fields. A `delete` button should also be available when editing an existing character.

The fields on our form will be as follows:

- ❑ Character name (alias)
- ❑ Real name

Building Databases

- Powers (multiple select field)
- Lair address, city, state, and ZIP
- Alignment (radio button: good/evil, default good)
- Enemies (multiple select field)

We also need a form for adding/deleting powers. This form will be relatively simple and will contain the following elements:

- Checkbox list of every power currently available
- Delete Selected button
- A text field to enter a new power
- An Add Power button

We also need a PHP script that can handle all database inserts, deletes, and so on. We call this a *transaction page*, and it simply does a required job and redirects the user on to another page. This page handles all transactions for the character application (with redirect), including the following:

- Inserting a new character (character listing page)
- Editing an existing character (character listing page)
- Deleting a character (character listing page)
- Adding a new power (power editor page)
- Deleting a power (power editor page)

That's basically all there is to the application. Four pages (five if you count the `config.php` file you created earlier—it will be used again) shouldn't be too difficult. Let's write them first, and then we'll talk about how they work.

Try It Out The Comic Book Character Site

Some of these files are a bit long. Don't let that scare you. Most of the code consists of SQL statements, and we explain them clearly for you in the "How It Works" section that follows. Remember that this code can also be downloaded from the Web site (www.wrox.com).

Let's start with a short one.

1. Enter the following code in your favorite PHP editor, and save it as `poweredit.php`:

```
<?php
require('config.php');

$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB, $conn);

$sql = "SELECT id, power FROM char_power ORDER BY power";
```

Chapter 9

```

$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        $pwrlist[$row['id']] = $row['power'];
    }
    $numpwr = count($pwrlist);
    $thresh = 5;
    $maxcols = 3;
    $cols = min($maxcols, (ceil(count($pwrlist)/$thresh)));
    $percol = ceil(count($pwrlist)/$cols);
    $powerchk = '';
    $i = 0;
    foreach ($pwrlist as $id => $pwr) {
        if (($i>0) && ($i%$percol == 0))
            $powerchk .= "</td>\n<td valign='top'>";
        $powerchk .= "<input type='checkbox' name='powers['
            value='$id'> $pwr<br />\n";
        $i++;
    }
    $delbutton = " <tr>
    <td colspan=\"$cols\" bgcolor=\"#CCCCCC\" align=\"center\">
    <input type=\"submit\" name=\"action\" value=\"Delete Powers\">
    <font size=\"2\" color=\"#990000\"><br /><br />
    deleting will remove all associated powers
    <br />from characters as well – select wisely</font>
    </td>
    </tr>";
} else {
    $powerchk = "<div style=\"text-align:center;width:300;
    font-family:Tahoma,Verdana,Arial\">No Powers entered...</div>";
}

?>
<html>
<head>
<title>Add/Delete Powers</title>
</head>
<body>

<h1>Comic Book<br />Appreciation</h1><br />
<h3>Editing Character Powers</h3>
<form action="char_transact.php" method="post" name="theform">
<table border="0" cellpadding="5">
<tr bgcolor="#FFCCCC">
<td valign="top"><?php echo $powerchk;?></td>
</tr>
<?php echo $delbutton; ?>
<tr>
<td colspan="<?php echo $cols;?>" bgcolor="#CCCCCC" align="center">
<input type="text" name="newpower" value="" size=20>
<input type="submit" name="action" value="Add Power">
</td>
</tr>
</table>

```

```

</form>
<a href="charlist.php">Return to Home Page</a>
</body>
</html>

```

2. Enter the following code, and save it as `charlist.php`:

```

<?php
require('config.php');

$ord = $_GET['o'];
if (is_numeric($ord)){
    $ord = round(min(max($ord, 1), 3));
} else {
    $ord = 1;
}
$order = array(
    1 => 'alias ASC',
    2 => 'name ASC',
    3 => 'align ASC, alias ASC'
);

$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB, $conn);

$sql = "SELECT c.id, p.power FROM char_main c JOIN char_power p JOIN
    char_power_link pk ON c.id = pk.char_id AND p.id = pk.power_id";

$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        $p[$row['id']][] = $row['power'];
    }
    foreach ($p as $key => $value) {
        $powers[$key] = implode(", ", $value);
    }
}

$sql = "SELECT c.id, n.alias FROM char_main c JOIN char_good_bad_link
    gb JOIN char_main n ON (c.id = gb.good_id AND n.id = gb.bad_id)
    OR (n.id = gb.good_id AND c.id = gb.bad_id)";

$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        $e[$row['id']][] = $row['alias'];
    }
    foreach ($e as $key => $value) {
        $enemies[$key] = implode(", ", $value);
    }
}
$table = "<table><tr><td align=\"center\">No characters currently
    exist.</td></tr></table>";

```

Chapter 9

```

?>

<html>
<head>
<title>Comic Book Appreciation</title>
</head>
<body>
<img src='CBA_Tiny.gif' align='left' hspace='10'>
<h1>Comic Book<br />Appreciation</h1><br />
<h3>Character Database</h3>

<?php
$sql = "SELECT id, alias, real_name AS name, align
      FROM char_main ORDER BY ". $order[$ord];

$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    $table = "<table border='0' cellpadding='5'>";
    $table .= "<tr bgcolor='#FFCCCC'><th>";
    $table .= "<a href='" . $_SERVER['PHP_SELF'] . "?o=1'>Alias</a>";
    $table .= "</th><th><a href='" . $_SERVER['PHP_SELF'] . "?o=2'>";
    $table .= "Name</a></th><th><a href='" . $_SERVER['PHP_SELF']";
    $table .= "?o=3'>Alignment</a></th><th>Powers</th>";
    $table .= "<th>Enemies</th></tr>";

    // build each table row
    while ($row = mysql_fetch_assoc($result)) {
        $bg = ($bg=='F2F2FF'? 'E2E2F2': 'F2F2FF');
        $pow = ($powers[$row['id']]=='?'none':$powers[$row['id']]);
        $ene = ($enemies[$row['id']]=='?'none':$enemies[$row['id']]);
        $table .= "<tr bgcolor='#" . $bg . "'><td><a href='charedit.php?c="
            . $row['id'] . "'> . $row['alias']. "</a></td><td>"
            . $row['name'] . "</td><td align='center'>" . $row['align']
            . "</td><td>" . $pow . "</td><td align='center'>" . $ene
            . "</td></tr>";
    }

    $table .= "</table>";
    $table = str_replace('evil', '<font color="red">evil</font>', $table);
    $table = str_replace('good', '<font color="darkgreen">good</font>',
        $table);
}
echo $table;
?>
<br /><a href="charedit.php">New Character</a> &bull;
<a href="poweredit.php">Edit Powers</a>
</body>
</html>

```

3. (Two down, two to go.) Enter the next block of code and save it as `charedit.php`:

```

<?php
require('config.php');

$char = $_GET['c'];

```

Building Databases

```

if ($char == '' || !is_numeric($char)) $char='0';
$subtype = "Create";
$subhead = "Please enter character data and click '$subtype
Character.'";
$stablebg = '#EEEEFF';

$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB,$conn);

$sql = "SELECT id, power FROM char_power";
$result = mysql_query($sql);
if (mysql_num_rows($result) > 0) {
  While ($row = mysql_fetch_assoc($result)) {
    $pwrlist[$row['id']] = $row['power'];
  }
}

$sql = "SELECT id, alias FROM char_main WHERE id != $char";
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
  $row = mysql_fetch_assoc($result);
  $charlist[$row['id']] = $row['alias'];
}

if ($char != '0') {
  $sql = "SELECT c.alias, c.real_name AS name, c.align, l.lair_addr
AS address, z.city, z.state, z.id AS zip FROM char_main c,
char_lair l, char_zipcode z WHERE z.id = l.zip_id AND
c.lair_id = l.id AND c.id = $char";
$result = mysql_query($sql) or die(mysql_error());
$ch = mysql_fetch_assoc($result);

if (is_array($ch)) {
  $subtype = "Update";
  $stablebg = '#EEFFEE';
  $subhead = "Edit data for <i>" . $ch['alias'] . "</i> and click
'$subtype Character.'";

  $sql = "SELECT p.id FROM char_main c JOIN char_power p
JOIN char_power_link pk ON c.id = pk.char_id
AND p.id = pk.power_id WHERE c.id = $char";
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
  While ($row = mysql_fetch_assoc($result)) {
    $powers[$row['id']] = 'selected';
  }
}

// get list of character's enemies
$sql = "SELECT n.id FROM char_main c JOIN char_good_bad_link gb
JOIN char_main n ON (c.id = gb.good_id AND n.id = gb.bad_id)
OR (n.id = gb.good_id AND c.id = gb.bad_id) WHERE
c.id = $char";
$result = mysql_query($sql) or die(mysql_error());

```

Chapter 9

```

if (mysql_num_rows($result) > 0) {
    While ($row = mysql_fetch_assoc($result)) {
        $enemies[$row['id']] = 'selected';
    }
}
}
}
?>

<html>
<head>
<title>Character Editor</title>
</head>
<body>
<img src='CBA_Tiny.gif' align='left' hspace='10'>
<h1>Comic Book<br />Appreciation</h1><br />
<h3><?php echo $subhead;?></h3>

<form action='char_transact.php' name='theform' method='post'>
<table border='0' cellpadding='15' bgcolor='<?php echo $tablebg;?>'>
<tr>
<td>Character Name:</td>
<td><input type='text' name='alias' size='41'
value='<?php echo $ch['alias'];?>'
>
</td>
</tr>
<tr>
<td>Real Name:</td>
<td><input type='text' name='name' size='41'
value='<?php echo $ch['name'];?>'
>
</td>
</tr>
<tr>
<td>Powers:<br /><font size=2 color='#990000'>
(Ctrl-click to<br />select multiple<br />powers)</font>
</td>
<td>
<select multiple='multiple' name='powers[]' size='4'>
<?php
foreach ($pwrlist as $key => $value) {
echo "<option value='$key' " . $powers[$key] .
"$value</option>\n";
}
?>
</select>
</td>
</tr>
<tr>
<td>Lair Location:<br /><font size=2 color='#990000'>
(address,<br />city, state, zip)</font>

```


Chapter 9

```
<input type='hidden' name='cid' value='<?php echo $char;?>'>
</form>
<a href='charlist.php'>Return to Home Page</a>
</body>
</html>
```

4. Okay, only one more now. This code is the longest, but that's because it contains a lot of SQL statements. It's not as bad as it looks. But if you want to download this code from the Web site, go ahead, and be guilt-free. Consider it our gift to you. If you are typing it, you know the drill. After entering it, save this one as `char_transact.php`:

```
<?php
require('config.php');
foreach($_POST as $key => $value) {
    $$key = $value;
}

$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB, $conn);

switch ($action) {
    case "Create Character":
        $sql = "INSERT IGNORE INTO char_zipcode (id, city, state)
            VALUES ('$zip', '$city', '$state')";
        $result = mysql_query($sql) or die(mysql_error());

        $sql = "INSERT INTO char_lair (id, zip_id, lair_addr)
            VALUES (NULL, '$zip', '$address')";
        $result = mysql_query($sql) or die(mysql_error());
        if ($result) $lairid = mysql_insert_id($conn);

        $sql = "INSERT INTO char_main (id, lair_id, alias, real_name, align)
            VALUES (NULL, '$lairid', '$alias', '$name', '$align')";
        $result = mysql_query($sql) or die(mysql_error());
        if ($result) $charid = mysql_insert_id($conn);

        if ($powers != "") {
            $val = "";
            foreach ($powers as $key => $id) {
                $val[] = "('$charid', '$id)";
            }
            $values = implode(',', $val);
            $sql = "INSERT IGNORE INTO char_power_link (char_id, power_id)
                VALUES $values";
            $result = mysql_query($sql) or die(mysql_error());
        }

        if ($enemies != '') {
            $val = "";
            foreach ($enemies as $key => $id) {
                $val[] = "('$charid', '$id)";
            }
        }
    }
}
```



```

    }
    $values = implode(',', $val);
    if ($align = 'good') {
        $cols = '(good_id, bad_id)';
    } else {
        $cols = '(bad_id, good_id)';
    }
    $sql = "INSERT IGNORE INTO char_good_bad_link $cols
    VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}

$redirect = 'charlist.php';
break;

case "Delete Character":
    $sql = "DELETE FROM char_main, char_lair USING char_main m,
    char_lair l WHERE m.lair_id = l.id AND m.id = $cid";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "DELETE FROM char_power_link WHERE char_id = $cid";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "DELETE FROM char_good_bad_link WHERE good_id = $cid
    OR bad_id = $cid";
    $result = mysql_query($sql) or die(mysql_error());

    $redirect = 'charlist.php';
    break;
case "Update Character":
    $sql = "INSERT IGNORE INTO char_zipcode (id, city, state)
    VALUES ('$zip', '$city', '$state')";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "UPDATE char_lair l, char_main m SET l.zip_id='$zip',
    l.lair_addr='$address', alias='$alias', real_name='$name',
    align='$align' WHERE m.id = $cid AND m.lair_id = l.id";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "DELETE FROM char_power_link WHERE char_id = $cid";
    $result = mysql_query($sql) or die(mysql_error());

    if ($powers != "") {
        $val = "";
        foreach ($powers as $key => $id) {
            $val[] = "('$cid', '$id)";
        }
        $values = implode(',', $val);
        $sql = "INSERT IGNORE INTO char_power_link (char_id, power_id)
        VALUES $values";
        $result = mysql_query($sql) or die(mysql_error());
    }

    $sql = "DELETE FROM char_good_bad_link WHERE good_id = $cid OR

```

Chapter 9

```
bad_id = $cid";
$result = mysql_query($sql) or die(mysql_error());

if ($enemies != '') {
    $val = "";
    foreach ($enemies as $key => $id) {
        $val[] = "('$cid', '$id')";
    }
    $values = implode(',', $val);
    if ($align == 'good') {
        $cols = '(good_id, bad_id)';
    } else {
        $cols = '(bad_id, good_id)';
    }
    $sql = "INSERT IGNORE INTO char_good_bad_link $cols
VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}

$redirect = 'charlist.php';
break;

case "Delete Powers":
if ($powers != "") {
    $powerlist = implode(',', $powers);

    $sql = "DELETE FROM char_power WHERE id IN ($powerlist)";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "DELETE FROM char_power_link
WHERE power_id IN ($powerlist)";
    $result = mysql_query($sql) or die(mysql_error());
}

$redirect = 'poweredit.php';
break;

case "Add Power":
if ($newpower != '') {
    $sql = "INSERT IGNORE INTO char_power (id, power)
VALUES (NULL, '$newpower')";
    $result = mysql_query($sql) or die(mysql_error());
}

$redirect = 'poweredit.php';
break;

default:

$redirect = 'charlist.php';
break;
}
header("Location: $redirect");
?>
```

Building Databases

Excellent work! We hope you typed everything correctly. (Remember, if your code is exactly as you see it in this chapter, and it doesn't work correctly, it's because we're testing your debugging and problem solving skills.)

Before we run through the code and figure out how it works, let's play with it a bit.

1. Open your browser, and point it to the location of `charlist.php`.

This is your Character Database home page. It should look something like Figure 9-1. If the logo is missing, you can download it from the Web site, edit the four pages to eliminate the image, or change it to anything you want. Because you don't currently have any characters to look at, let's move on.

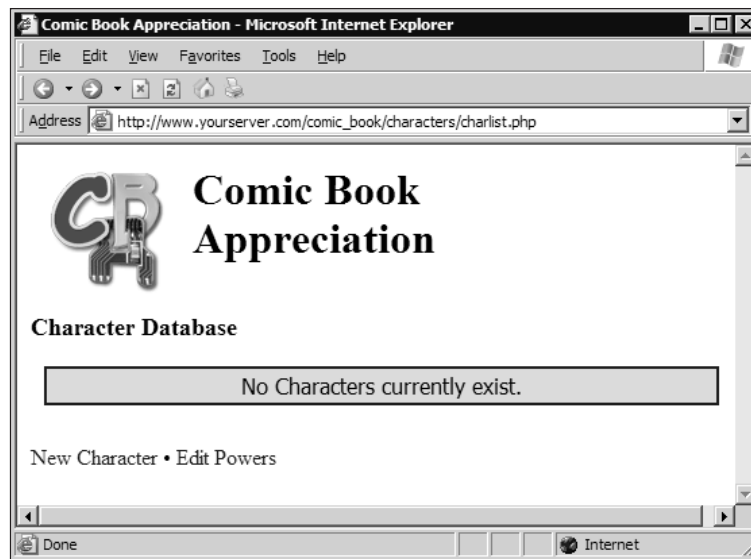


Figure 9-1

2. Click "Edit Powers."

When the page appears (see Figure 9-2), it initially will be empty.
3. Enter an ultra-cool superpower such as invisibility or x-ray vision in the text box, and click Add Power.

If you need help with power ideas, here are a few: super strength, invisibility, x-ray vision, speed, soccer mom, stretchable, flight, breathes underwater. Add a total of six powers. Moving on . . .

You should now see a new button and a list of powers with checkboxes next to them.
4. Check one or two powers and click Delete Powers. They should go away.
5. When you finish editing the powers, click the link at the bottom, "Return to Home Page," which takes you back to the Character List page (which of course still has no characters).

Chapter 9



Figure 9-2

6. Click the link New Character.

A shiny new page appears (with a blue background), ready for your data input (see Figure 9-3). You will notice that the powers you entered are now choices in the Powers field. (Relational databases rule!)

7. Enter the appropriate data, and click Create Character.

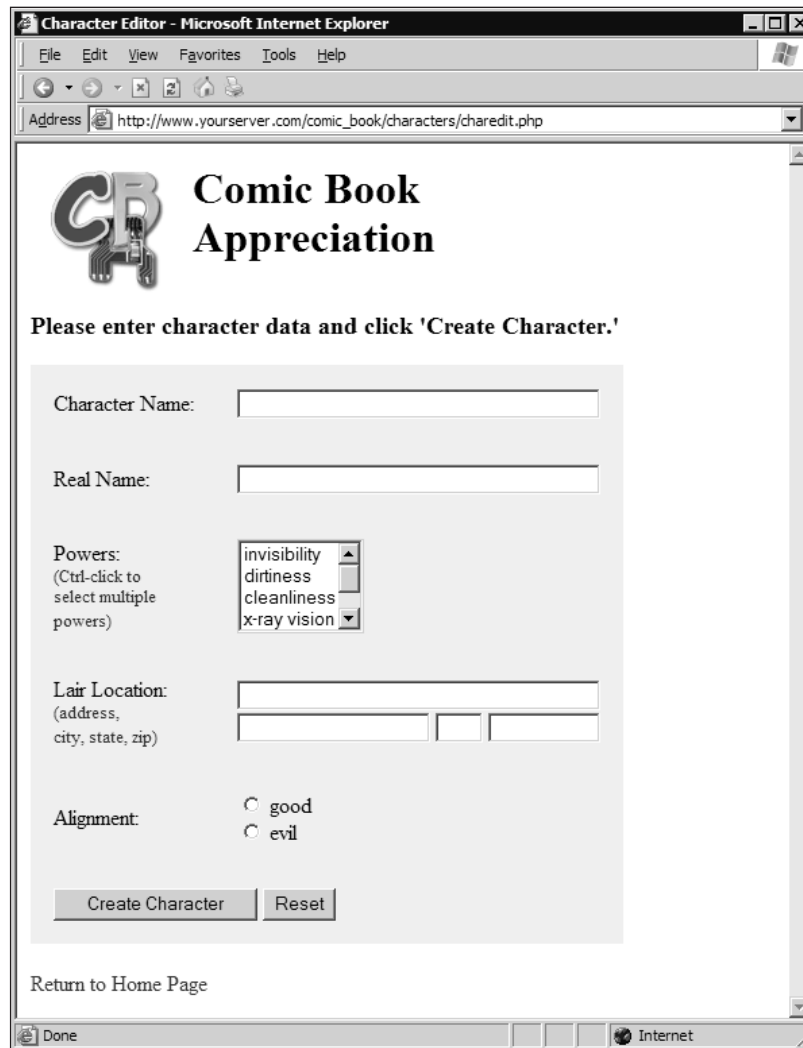
You should be taken to the home page, where you'll now see the character you entered (as in Figure 9-4).

8. If you click New Character again, you should now see an extra field for Enemies. You can select any previously created character in the database as the current character's enemy.
9. From the home page, click one of your characters' names.

The Character Editor page loads again, but now the background is green, and the character's data will be automatically entered into the fields (see Figure 9-5). If you look at the URL for this page, you see `?c=x` at the end, where `x` is the character's number.

10. Change some of the data, and click Update Character.

You are taken back to the home page, and you should immediately see the results of your changes. In fact, if you selected an enemy for this character, you should see the results change in the enemy's row as well.



The screenshot shows a Microsoft Internet Explorer window titled "Character Editor - Microsoft Internet Explorer". The address bar displays "http://www.yourserver.com/comic_book/characters/charedit.php". The main content area features a logo with the letters "CB" and a character's head, followed by the heading "Comic Book Appreciation". Below the heading is the instruction "Please enter character data and click 'Create Character.'". The form includes several input fields: "Character Name:" (a single text box), "Real Name:" (a single text box), "Powers:" (a list box with "invisibility", "dirtiness", "cleanliness", and "x-ray vision" selected, and a note "(Ctrl-click to select multiple powers)"), "Lair Location:" (a text box for the address, and three smaller text boxes for city, state, and zip), and "Alignment:" (radio buttons for "good" and "evil"). At the bottom of the form are "Create Character" and "Reset" buttons. Below the form is a link "Return to Home Page". The browser's status bar at the bottom shows "Done" and "Internet".

Figure 9-3

Are you starting to see the benefits of relational databases? Are you ready to get under the hood and figure out what you just typed in those 499 (yes, 499!) lines of code? Let's go.

How It Works

Let's start off with `poweredit.php`. It's not too long, and it will allow us to get our feet wet with SQL, PHP, and HTML.

Chapter 9

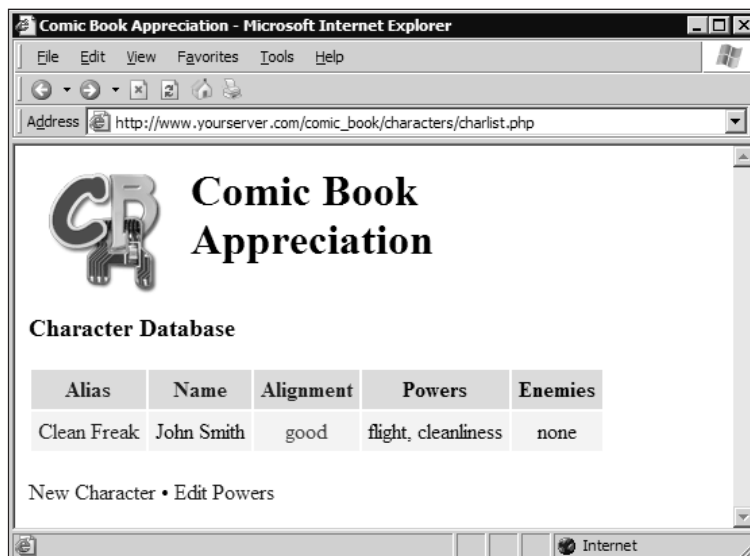


Figure 9-4

poweredit.php

You will see this on every page, but we will mention it this one time only. We include the `config.php` file that contains the constants used in the next couple of lines. By putting these constants in an included file, we can make any required changes in one place. We use the `require` command instead of `include` because of the way PHP works: An included file will not stop the processing of the rest of the page, whereas a required file, if not found, would immediately stop processing.

```
require('config.php');
```

Next, a connection to the server is made, and the appropriate database is selected. Notice the use of the constants we defined in `config.php`:

```
$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB, $conn);
```

What follows is a somewhat simple SQL select statement. It is grabbing the `id` and `power` columns from the `char_power` table, and sorting them by power. This way when we iterate through them later and put the data on the Web page, they will be in alphabetical order.

```
$sql = "SELECT id, power FROM char_power ORDER BY power";
```

This executes the SQL statement and throws an error if there are any problems:

```
$result = mysql_query($sql) or die(mysql_error());
```

The screenshot shows a web browser window titled "Character Editor - Microsoft Internet Explorer". The address bar contains the URL "http://www.yourserver.com/comic_book/characters/charedit.php?c=13". The page content includes a logo with the letters "CB" and the title "Comic Book Appreciation". Below the title is the instruction "Edit data for *Clean Freak* and click 'Update Character.'". The form contains the following fields and options:

- Character Name:** Clean Freak
- Real Name:** John Smith
- Powers:** (Ctrl-click to select multiple powers)
 - invisibility
 - dirtiness
 - cleanliness** (highlighted)
 - x-ray vision
- Lair Location:** (address, city, state, zip)
 - 123 Poplar Ave.
 - Townsburg OH 45293
- Alignment:**
 - good
 - evil

At the bottom of the form are three buttons: "Update Character", "Reset", and "Delete Character". Below the form is a link "Return to Home Page". The browser status bar shows "Done" and "Internet".

Figure 9-5

Now we check to make sure at least one row was returned. If so, we iterate through each row, building up an array of powers, using the power id as the array key. Note the use of `mysql_fetch_assoc`. Other options are `mysql_fetch_row` and `mysql_fetch_array`. Because we only need an associative array (which uses field names as keys instead of a numerical index), `mysql_fetch_assoc` works nicely.

```
if (mysql_num_rows($result) > 0) {
    While ($row = mysql_fetch_assoc($result)) {
        $pwrlist[$row['id']] = $row['power'];
    }
}
```

Chapter 9

When retrieving data from the database, we will usually need to retrieve appropriate ids so that we can later insert or update the correct record. In this case, the id serves as the key to the array, making it easy to retrieve the values. We could have certainly used a multi-value array, but that gets a little more confusing, and it's just not necessary here. Just be sure you understand that many times in this application (and many apps using relational databases) you will use the table id as an array key.

Now we're going to get a little tricky. Because our list of powers could get quite large, we want to try to distribute them across multiple columns. However, we'd like to distribute them fairly evenly. The following 13 lines of code do this for us (if math is not interesting to you at all, or you simply don't want to know how this part of the code works, skip this section).

First, we get a count of the number of powers in our array. Next, we set the threshold to 5 lines (after which a second column will be created), and a maximum number of columns (in this case, 3).

```
$numpwr = count($pwrlist);
$thresh = 5;
$maxcols = 3;
```

Next, we determine how many columns to create. Let's assume there are 7 powers to display. First, we divide the count by the threshold (7/5), which gives us 1.4. Next, we use `ceil()` to round up to the nearest integer (`ceil(1.4) = 2`). Then we take the smaller of the two values (3 and 2), and store it in the `$cols` variable. In this example, `$cols` would equal 2.

To figure out how many powers go into each column, we divide the count by the number of columns, and round up to the nearest integer. In this case, `ceil(7/2) = 4`. So, we'll have two columns, with four values in each column (the last column will contain the remainder of powers if there are less than four).

```
$powerchk is a string that will contain each power, with a checkbox attached to it.
For now, we initialize it to an empty string ''. $cols = min($maxcols,
(ceil(count($pwrlist)/$thresh)));
$percol = ceil(count($pwrlist)/$cols);
$powerchk = '';
```

Now we loop through each element of the `$pwrlist` array, which contains the id as the key (`$id`), and power as the value (`$pwr`). Our counter `$i` will start at 0 and increment each time through the loop. In each loop, we add the `<input>` tag to create the checkbox, using the id as the value, and the name of the power as the label. When our counter reaches a value that is divisible by `$percol`, we add a close table definition and start a new one.

```
$i = 0;
foreach ($pwrlist as $id => $pwr) {
  if (($i>0) && ($i%$percol == 0))
    $powerchk .= "</td>\n<td valign='top'>";
  $powerchk .= "<input type='checkbox' name='powers[]'
    value='$id'> $pwr<br />\n";
  $i++;
}
```

In our example, increments 0, 1, 2, and 3 end up in the first column. When `$i` reaches 4 (the value of `$percol`), we start a new column. If this is confusing, don't worry. You can play around with it by

Building Databases

changing your `$thresh` and `$maxcols` values, and adding a bunch of random power values to see how the table is built. For now, let's check out the rest of the code.

This is the rest of our `if` loop. If there is even one power, a row is created that contains a delete button. If not, we create a row that simply states that no powers have yet been entered.

```
$delbutton = " <tr>
  <td colspan='$cols' bgcolor='#CCCCFF' align='center'>
    <input type='submit' name='action' value='Delete Powers'>
    <font size='2' color='#990000'><br /><br />
    deleting will remove all associated powers
    <br />from characters as well - select wisely</font>
  </td>
</tr>";
} else {
  $powerchk = "<div style='text-align:center;width:300;
  font-family:Tahoma,Verdana,Arial'>No Powers entered...</div>";
}
?>
```

We have left off some of the HTML. We assume you know HTML well enough that we don't need to explain it. As you can see in the `<form>` tag, when the user clicks the Add Power or Delete Powers button, we'll be sending values to `char_transact.php`:

```
<form action='char_transact.php' method='post' name='theform'>
```

At this point, `$powerchk` either contains the No Powers display, or the built up table columns. Either way, we insert `$powerchk` into the table. Note the open and close table definitions (`<td valign="top">` and `</td>`). We didn't add them to `$powerchk` earlier, but we *did* add the internal close/open definitions to create the columns as necessary.

```
<table border='0' cellpadding='5'>
  <tr bgcolor='#FFCCCC'>
    <td valign='top'><?php echo $powerchk;?></td>
```

In the following, `$delbutton` either contains the row with the delete button (if powers were found), or it's blank. That is how we control when it shows up, and this is where it's inserted into the table.

```
<?php echo $delbutton; ?>
```

The following deals with the add button. Notice that it is called 'action' and that it has a value of Add Power. When submitting a form, PHP passes these values on to the next page. Because we are using the post method on our form, we will have a `$_POST` variable called 'action' that contains the value of the button. Because of this, and because all of our forms load `char_transact.php`, all of our buttons are named 'action', and have different values so that we can determine what to do with the data that is sent. We go into more detail about this when we look at `char_transact.php`.

```
<input type='submit' name='action' value='Add Power'>
```

Chapter 9

charlist.php

The `charlist.php` page has an optional parameter that can be passed: `?o=x`, where `x` is 1, 2, or 3. This code retrieves that variable if it exists, and converts it to the appropriate value if necessary. If some smart-alec types `o=4` in the browser, the code return 3. If no value or a bad value is passed, it will default to 1. The value is stored in `$ord`.

```
$ord = $_GET['o'];
if (is_numeric($ord)){
    $ord = round(min(max($ord, 1), 3));
} else {
    $ord = 1;
}
$order = array(
    1 => 'alias ASC',
    2 => 'name ASC',
    3 => 'align ASC, alias ASC'
);
```

This value determines which column our character display will be sorted on: 1 is by alias, 2 is by real name, and 3 is by alignment and then alias. We will use the value `$ord` as the key to our order array, which will be appended to the appropriate SQL statement later.

Make a connection, and choose a database. You know the drill by now.

```
$conn = mysql_connect(SQL_HOST, SQL_USER, SQL_PASS)
    or die('Could not connect to MySQL database. ' . mysql_error());
mysql_select_db(SQL_DB, $conn);
```

Ah... our first JOIN. This `select` statement might look confusing to the uninitiated, but it is not that complicated. First, let's look at the JOIN statements. We are joining three tables, using the `char_power_link` table to link the `char_power` table and the `char_main` table. This is a many-to-many (M:N) relationship. We define how they are joined with the ON statement. As you can see, we are linking up the character table to the link table using the character id, and we're linking the power table to the link table using the power id. With that link established, you can see that we are grabbing the character's id and the powers assigned to each character.

```
$sql = "SELECT c.id, p.power FROM char_main c JOIN char_power p JOIN
    char_power_link pk ON c.id = pk.char_id AND p.id = pk.power_id";
$result = mysql_query($sql) or die(mysql_error());
```

Notice our use of aliases for the tables. The character table is `c`, the power link table is `pk`, and the power table is `p`. This allows us to refer to the appropriate columns with a shorter syntax (for example `pk.char_id` instead of `char_power_link.char_id`). It is not necessary to use `table.column` syntax if the column name is unique across all tables. However, it is a good practice to keep so that you are always aware of which data you are accessing. It is required, of course, for column names that are duplicated across multiple tables (such as `id`). Some might recommend that you *always* use unique names for all of your fields, but we prefer the practice of naming all primary keys "id" and using proper `table.column` syntax in our SQL queries.

Building Databases

Next, we are creating a multidimensional array. That's fancy talk for an array with more than one index. This one is two-dimensional. Think of a two-dimensional array as being like a spreadsheet, and it isn't difficult to understand.

```
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        $p[$row['id']][] = $row['power'];
    }
}
```

The trick here is that we have multiple powers for the same id. By adding [] to the \$p array, a new array item is created for each row that has the same id. The end result is that you have a \$p array of *x* characters, each element of which contains a \$p[*x*] array of *y* powers. That is a multidimensional array.

Now we go back through our temporary array \$p, and pull out each array that it holds. The \$key variable contains the character id, and \$value contains the array of that character's powers. We then implode the powers into a comma-separated list of powers, and store that in the \$powers array, using the character id (\$key) as the array index. We end up with an array that contains a list of powers for each character.

```
foreach ($p as $key => $value) {
    $powers[$key] = implode(", ", $value);
}
```

Oh boy, another JOIN. This one is similar to the previous M:N query, with a couple of exceptions. First of all, we are linking the character table twice. You can see that we are creating two instances of that table, one called c for "character" and one called n for "nemesis." This distinction is very important.

```
$sql = "SELECT c.id, n.alias FROM char_main c JOIN char_good_bad_link
gb JOIN char_main n ON (c.id = gb.good_id AND n.id = gb.bad_id)
OR (n.id = gb.good_id AND c.id = gb.bad_id)";
```

The other exception is the ON statement. We have characters that we are attempting to link to other characters as "enemies." Call them opponents, or nemesis, or whatever. Typically, you expect good versus evil and vice-versa. However, we are allowing *any* character to be the enemy of *any other* character. That makes linking more interesting because we are using a table with a bad_id and a good_id. If you have two evil characters that are enemies, which one gets stored in the good_id column?

The answer is that it doesn't matter. What we want to do is to make sure that we not only don't have any duplicates in the char_good_bad_link table, but also that we don't have what we call *reverse duplication*. In other words, if you have a row with good_id=3 and bad_id=7, then good_id=7 and bad_id=3 must be considered a duplicate. There is no way to prevent that in MySQL using primary keys, so we must take care of that contingency in our code. We do that in a couple of places.

In this instance, we are combining two queries in one. The first one grabs all instances of each character where the character's id is in the good_id field and his enemies' IDs are in the bad_id field. The second part of the ON statement reverses that, and pulls all instances of each character where the character's ID is in the bad_id field and his enemies' ids are in the good_id field. This does not prevent reverse duplication (that is handled elsewhere), but it does make sure we have grabbed every possible link to a character's enemy.

Chapter 9

This code is virtually identical to the multidimensional powers array. This time, we are creating a multidimensional array of each character and that character's enemies. We then implode the enemies list and store it in the `$enemies` array, using the character's id as the array index.

```
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        $e[$row['id']][] = $row['alias'];
    }
    foreach ($e as $key => $value) {
        $enemies[$key] = implode(", ", $value);
    }
}
```

We are going to build a table of characters in a moment. In case there are no characters to display (as when you first tested your `charlist.php` page), we want to display a "No characters" message. This code builds the `$table` variable (even though it doesn't contain an actual table) using a `<div>` tag. If any characters do exist, this variable will be overwritten with an actual table of data.

```
$table = "<table><tr><td align=\"center\">No characters currently
        exist.</td></tr></table>"

?>
```

Next is another simple SQL `SELECT`, pulling the appropriate data: character's id, alias, real name, alignment, and address info. Note the `$order` array. We set that value at the beginning of this page, using the `?_GET` value "o" in the URL. This is where it's used to sort the characters.

```
$sql = "SELECT id, alias, real_name AS name, align
        FROM char_main ORDER BY ". $order[$ord];
$result = mysql_query($sql) or die(mysql_error());
```

We are building up the table of characters, as long as we returned at least one record from the database. Note the first three columns' links. They refer back to this same page, adding the `?o=x` parameter. This will re-sort the data and display it sorted on the column the user clicked.

```
if (mysql_num_rows($result) > 0) {
    $table = "<table border='0' cellpadding='5'>";
    $table .= "<tr bgcolor='#FFCCCC'><th>";
    $table .= "<a href=\"" . $_SERVER['PHP_SELF'] . "?o=1'>Alias</a>";
    $table .= "</th><th><a href=\"" . $_SERVER['PHP_SELF'] . "?o=2'>";
    $table .= "Name</a></th><th><a href=\"" . $_SERVER['PHP_SELF'] . "?o=3'>Alignment</a></th><th>Powers</th>";
    $table .= "<th>Enemies</th></tr>";
```

Next, we alternate the background colors of the table, to make it a little easier to read.

```
// build each table row
while ($row = mysql_fetch_assoc($result)) {
    $bg = ($bg=='F2F2FF'? 'E2E2F2': 'F2F2FF');
```

Building Databases

Remember the power and enemy arrays we built earlier? We use the character's id to grab the list of values and put them into a variable to be inserted shortly into the appropriate table cell.

```
$pow = ($powers[$row['id']]=='?'?none:$powers[$row['id']]);
$ene = ($enemies[$row['id']]=='?'?none:$enemies[$row['id']]);
```

The table is built, row by row, inserting the appropriate data in each cell; then it's closed:

```
$table .= "<tr bgcolor='#" . $bg . "'><td><a href='charedit.php?c="
. $row['id'] . "'>" . $row['alias'] . "</a></td><td>"
. $row['name'] . "</td><td align='center'>" . $row['align']
. "</td><td>" . $pow . "</td><td align='center'>" . $ene
. "</td></tr>";
$table .= "</table>";
```

Just for kicks, and to make them more visible, we change the color of the “good” and “evil” values in the table. This isn't necessary, but it makes the values pop out more.

```
$table = str_replace('evil', '<font color="red">evil</font>', $table);
$table = str_replace('good', '<font color="darkgreen">good</font>',
    $table);
```

This variable contains either the <div> tag we created earlier or the table of character data. It's inserted in the page here.

```
echo $table;
```

charedit.php

This file does double-duty, so it's a little longer. But a lot of it is HTML, and much of what it does we have already done before, so this shouldn't be too difficult.

The default functionality of this page is New Character mode. If there is a value in \$char other than 0, we will pull the data and change the default values.

```
$char = $_GET['c'];
if ($char == '' || !is_numeric($char)) $char='0';
$subtype = "Create";
$subhead = "Please enter character data and click '$subtype
    Character.'";
$tablebg = '#EEEEFF';
```

Get all powers, and put them into an array to be accessed later (when building the power select field on the form).

```
$sql = "SELECT id, power FROM char_power";
$result = mysql_query($sql);
if (mysql_num_rows($result) > 0) {
    While ($row = mysql_fetch_assoc($result)) {
        $pwrlist[$row['id']] = $row['power'];
    }
}
```

Chapter 9

All characters except the chosen character will be pulled from the database to be used for the Enemies field. If the character id is not valid, then *all* characters will be pulled for the Enemies field.

```
$sql = "SELECT id, alias FROM char_main WHERE id != $char";
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    $row = mysql_fetch_assoc($result);
    $charlist[$row['id']] = $row['alias'];
}
```

If there is a character id, attempt to pull the data from the database. This SQL statement is also a JOIN, although the JOIN keyword is not used. You can identify such a JOIN because there are two or more tables, and the WHERE keyword is matching columns from each of the tables. The JOIN in this case is implied. Once all the tables are joined, all the appropriate fields are pulled as long as the character id in the character table matches \$char. If there is no match, no records will be returned. If there is a match, one record is returned and the row is stored in \$ch.

```
if ($char != '0') {
    $sql = "SELECT c.alias, c.real_name AS name, c.align, l.lair_addr
        AS address, z.city, z.state, z.id AS zip FROM char_main c,
        char_lair l, char_zipcode z WHERE z.id = l.zip_id AND
        c.lair_id = l.id AND c.id = $char";
    $result = mysql_query($sql) or die(mysql_error());
    $ch = mysql_fetch_assoc($result);
}
```

Once we determine there was a record retrieved, we alter the default variables to reflect the edited document. The background is green, and we are “Updating” rather than “Creating.”

```
if (is_array($ch)) {
    $subtype = "Update";
    $tablebg = '#EEFFEE';
    $subhead = "Edit data for <i>" . $ch['alias'] . "</i> and click
        '$subtype Character.'";
}
```

The next SQL statement retrieves all powers associated with this character. All we really need is the id so that we can create a \$powers array with each element containing the word “selected.” This will be used in the Powers field on the form, so that each power assigned to the character will be automatically selected.

```
$sql = "SELECT p.id FROM char_main c JOIN char_power p
    JOIN char_power_link pk ON c.id = pk.char_id
    AND p.id = pk.power_id WHERE c.id = $char";
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    While ($row = mysql_fetch_assoc($result)) {
        $powers[$row['id']] = 'selected';
    }
}
```

Now we do exactly the same thing with the character’s enemies. Note the similarity in this SQL statement to the one in charlist.php. The only difference is that we want only the enemies that match our character.

Building Databases

```
// get list of character's enemies
$sql = "SELECT n.id FROM char_main c JOIN char_good_bad_link gb
      JOIN char_main n ON (c.id = gb.good_id AND n.id = gb.bad_id)
      OR (n.id = gb.good_id AND c.id = gb.bad_id) WHERE
      c.id = $char";
$result = mysql_query($sql) or die(mysql_error());
if (mysql_num_rows($result) > 0) {
    While ($row = mysql_fetch_assoc($result)) {
        $enemies[$row['id']] = 'selected';
    }
}
```

We next build the table in HTML, and insert values into the appropriate places as defaults. This is how we fill in the fields with character data. Note the use of the PHP tag. We don't recommend using the shortcut `<?=$variable?>` because some servers don't have short PHP tags enabled. Using the full syntax guarantees that your code will work, regardless of what server it is on.

```
<td>Character Name:</td>
<td><input type='text' name='alias' size='41'
      value='<?php echo $ch['alias'];?>'
      onfocus='this.select();'>
</td>
```

Now we build the Powers select field. As we loop through each power in the `$pwrlist` array (which contains *all* powers), we concatenate the `$powers` array value for that power ("selected"). If that power's key (from `$pwrlist`) doesn't exist in the `$powers` array, `$powers[$key]` will simply be blank instead of "selected." In this way, we build a field of *all* powers where the character's chosen powers are selected in the list. Neato, huh?

```
<td>Powers:<br /><font size=2 color='#990000'>
      (Ctrl-click to<br />select multiple<br />powers)</font>
</td>
<td>
<select multiple='multiple' name='powers[]' size='4'>
<?php
      foreach ($pwrlist as $key => $value) {
          echo "    <option value='$key' " . $powers[$key] .
              ">$value</option>\n";
      }
?>
</select>
</td>
```

Note the `[]` in the select *name* attribute. That is necessary for PHP to recognize the variable as an array when it gets POSTed to the next page. This is a requirement for any field that might post with multiple values.

The following portion creates a set of radio buttons for "good" and "evil." The character's alignment is selected with the checked attribute.

Building Databases

You should always assume that `register_globals` is turned OFF to make your application more portable, and for this reason, we assume that we have access to the posted variables through the `$_POST` array only. What we are doing here is looping through `$_POST` and setting each variable ourselves. If `username` was passed as `$_POST['username']`, then it will now be accessible as `$username`, regardless of the `register_globals` setting.

```
foreach($_POST as $key => $value) {
    $$key = $value;
}
```

Remember that each button is named `action` and that each one has a different value. In the code that follows, we determine which button was clicked, and run the appropriate code. For example, if the Delete Character button was clicked, we want to run the SQL commands only for removing character data.

```
switch ($action) {
```

The `switch` command is a fancy way of providing a multiple choice. It is easier to read than a complex `if...else` statement. The only “gotcha” you need to be aware of is to use `break`; at the end of each case to prevent the rest of the code in the other case blocks from executing.

The `INSERT` query that follows is relatively simple. In plain English: “Insert the values `$zip`, `$city`, and `$state` into the columns `id`, `city`, and `state` in the `char_zipcode` table.” The `IGNORE` keyword is a very cool option that allows you do an insert without first using a `SELECT` query to see if the data is already in the table. In this case, you know there might already be a record for this Zip code. So, `IGNORE` tells the query “If you see this Zip code in the database already, don’t do the `INSERT`.”

```
case "Create Character":
    $sql = "INSERT IGNORE INTO char_zipcode (id, city, state)
    VALUES ('$zip', '$city', '$state')";
    $result = mysql_query($sql) or die(mysql_error());
```

Note that the `IGNORE` statement compares primary keys only. Therefore, even if another Zip code is in the database with the same state, the `INSERT` still takes place. Using `IGNORE` when inserting into a table where the primary key is automatically incremented has no effect at all; the `INSERT` will *always* happen in that case. This might seem obvious to you, but just keep this fact in mind; with some complex tables it won’t be so intuitive.

In the `INSERT` that follows, you see the use of `NULL` as the first value. When you insert `NULL` into a column, MySQL does the following: If the column allows `NULL` values, it inserts the `NULL`; if it does not allow `NULL` (the column is set to `NOT NULL`), it will set the column to the default value. If a default value has not been determined, then the standard default for the datatype is inserted (empty string for `var-char/char`, 0 for integer, and so on). If, as is the case here, the column is set to `auto_increment`, then the next highest available integer for that column is inserted. In our case, `id` is the primary key, so this is what we want to happen.

```
$sql = "INSERT INTO char_lair (id, zip_id, lair_addr)
VALUES (NULL, '$zip', '$address')";
$result = mysql_query($sql) or die(mysql_error());
```

Chapter 9

We also could have left out the `id` field from the insert, and inserted values into the `zip_id` and `lair_addr` columns only. MySQL treats ignored columns as if you had attempted to insert `NULL` into them. We like to specify every column when doing an insert. If you needed to modify your SQL statement later, having all the columns in the `INSERT` query gives you a nice placeholder so all you have to do is modify the inserted value.

The following is a neat little function. Assuming the insert worked properly (`$result` returned `TRUE`), the `mysql_insert_id()` function will return the value of the last `auto_increment` from the last run query. This works only after running a query on a table with an `auto_incremented` column. In this case it returns the primary key value of the row we just inserted into the `char_lair` table. We will need that value to insert into the `char_main` table.

```
if ($result) $lairid = mysql_insert_id($conn);
```

The connection variable is optional, but we think it's a good habit to always include it. If you omit it, it will use the most recently opened connection. In a simple application like ours, that's not a problem; in a more complex application where you might have more than one connection, it could get confusing.

Again, note the use of `NULL` for the primary key `id`, and the use of `mysql_insert_id()` to return the primary key in the following:

```
$sql = "INSERT INTO char_main (id, lair_id, alias, real_name, align)
VALUES (NULL, '$lairid', '$alias', '$name', '$align)";
$result = mysql_query($sql) or die(mysql_error());
if ($result) $charid = mysql_insert_id($conn);
```

We are always interested in minimizing the number of times we run a query on the database. Each hit takes precious time, which can be noticeable in a more complex application. At this point, we need to figure out how to insert all powers with only one SQL command:

```
if ($powers != "") {
    $val = "";
    foreach ($powers as $key => $id) {
        $val[] = "('$charid', '$id)";
    }
    $values = implode(', ', $val);
    $sql = "INSERT IGNORE INTO char_power_link (char_id, power_id)
VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}
```

There are a couple of concerns here. First, if there is already a power for this user (there shouldn't be; it's a new character, but always be prepared), we need to not insert the row. We already know how to take care of this by using the `IGNORE` keyword.

Second, we must insert multiple rows of data with only one query. Easy enough; all we have to do is supply a comma-separated list of value groupings that matches up to the column grouping in the query. For example:

```
INSERT INTO table (col1, col2) VALUES (val1, val2), (val3, val4)
```

Building Databases

We accomplish this by looping through the `$powers` array and putting the values for character id and power id into a new array. We then concatenate that array with a comma separator, and *voilà!* There are your multiple rows of data to insert.

We then do the same thing with the `$enemies` array that we did with `$powers`. This time, however, we insert into the columns based on whether the character is good or evil. It doesn't really matter too much which column gets which id, but for the most part we want evil character ids in the `bad_id` column.

```
if ($enemies != '') {
    $val = "";
    foreach ($enemies as $key => $id) {
        $val[] = ("'$charid', '$id'");
    }
    $values = implode(',', $val);
    if ($align = 'good') {
        $cols = '(good_id, bad_id)';
    } else {
        $cols = '(bad_id, good_id)';
    }
    $sql = "INSERT IGNORE INTO char_good_bad_link $cols
        VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}
```

When it comes to the `char_good_bad_link` table, we have a little bit of referential integrity that we have to handle (beyond what MySQL does for us). Namely, we don't want to have a `good_id/bad_id` combination to match up to a `bad_id/good_id` combination. For the purposes of a relational database, that isn't bad, but for our purposes that is considered a duplication. We will handle this contingency when updating a character, but because this is a new character (with a brand new id), we don't have to worry about that just yet.

We're done inserting new character data, so we now set the page we are going to load next, and break out of the `switch` statement.

```
$redirect = 'charlist.php';
break;
```

When deleting a character, we simply remove all instances of it from all relevant tables. In order to remove the relevant data from the `char_lair` table, we have to `JOIN` it to the `char_main` table by matching up the lair id's first. Then we delete all matching rows where the character id matches.

```
case "Delete Character":
    $sql = "DELETE FROM char_main, char_lair USING char_main m,
        char_lair l WHERE m.lair_id = l.id AND m.id = $cid";
    $result = mysql_query($sql) or die(mysql_error());

    $sql = "DELETE FROM char_power_link WHERE char_id = $cid";
    $result = mysql_query($sql) or die(mysql_error());
```

We don't really need to put the results of the `mysql_query` command in a variable. We like to do this as a matter of habit because if we ever need the return value later, it will be available for us to use. In the case of a `DELETE`, you don't get a result set, you get a return value of either `TRUE` or `FALSE`.

Chapter 9

Remembering that our `char_good_bad_link` needs to maintain what we call “reverse” referential integrity (1, 3 matches 3, 1), we remove all rows that contain the character’s id in either column:

```
$sql = "DELETE FROM char_good_bad_link WHERE good_id = $cid
OR bad_id = $cid";
$result = mysql_query($sql) or die(mysql_error());
```

Updating a character is where things get interesting. First of all, we can simply do an `INSERT IGNORE` on the Zip code table. If the address and Zip code change, we don’t really need to delete the old data because it might be used for other characters—it’s perfectly fine to leave the old data alone. So, we just do an `INSERT IGNORE` as we did for a new character, and leave it at that.

```
case "Update Character":
    $sql = "INSERT IGNORE INTO char_zipcode (id, city, state)
VALUES ('$zip', '$city', '$state')";
    $result = mysql_query($sql) or die(mysql_error());
```

Here is our first `UPDATE` query, and incidentally, the only one we use in the entire application. It is very similar to `INSERT` and `SELECT` queries, with the exception of the `SET` keyword. The `SET` keyword tells MySQL what columns to set, and what values to set them to. The old values in the row are overwritten. This is a `JOIN` query because there is more than one table. The `WHERE` keyword specifies both the linking column (`lair_id`) and the condition that only rows for this character will be updated.

```
$sql = "UPDATE char_lair l, char_main m SET l.zip_id='$zip',
l.lair_addr='$address', alias='$alias', real_name='$name',
align='$align' WHERE m.id = $cid AND m.lair_id = l.id";
$result = mysql_query($sql) or die(mysql_error());
```

Because the `char_power_link` table does not have an automatically incremented column as the primary key, we don’t have to do an update to the table. An update is possible, but it is much easier to simply delete all the old links of character to power, and insert new link rows. In some cases, we may be deleting and inserting the same data (for instance, we might be adding `flight` as a power, but `invisibility` did not change; `invisibility` will still be deleted and reinserted). When updating data in an M:N relationship, you will usually simply delete the old data, and insert the updated/new data.

```
$sql = "DELETE FROM char_power_link WHERE char_id = $cid";
$result = mysql_query($sql) or die(mysql_error());

if ($powers != "") {
    $val = "";
    foreach ($powers as $key => $id) {
        $val[] = "('$cid', '$id')";
    }
    $values = implode(', ', $val);
    $sql = "INSERT IGNORE INTO char_power_link (char_id, power_id)
VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}
```

Building Databases

This brings us to the Enemies data, where not only do we have to maintain referential integrity, but we have to worry about updating rows where our id can be present in either of the two linking columns. We must maintain our own “reverse” referential integrity.

```
$sql = "DELETE FROM char_good_bad_link WHERE good_id = $cid OR
bad_id = $cid";
$result = mysql_query($sql) or die(mysql_error());

if ($enemies != '') {
    $val = "";
    foreach ($enemies as $key => $id) {
        $val[] = "('$cid', '$id')";
    }
    $values = implode(',', $val);
    if ($align == 'good') {
        $cols = '(good_id, bad_id)';
    } else {
        $cols = '(bad_id, good_id)';
    }
    $sql = "INSERT IGNORE INTO char_good_bad_link $cols
VALUES $values";
    $result = mysql_query($sql) or die(mysql_error());
}
```

How did we deal with referential integrity? It turns out that it takes care of itself when we follow the same method we employed when updating the `char_power_link` table. By simply running the same `DELETE` query we ran when deleting a character, and then immediately running the same `INSERT` query we ran when creating a new character, we ensure that only one set of rows exists to match up each character to his/her enemy. It's simple, elegant, and it works!

By this time, queries should seem quite familiar to you. The `DELETE` query is one of the simplest of the SQL statements. In these `DELETE` queries, we need to delete each power that was selected on the Add/Delete Power page. We must do this not only in the `char_power` table but in the `char_power_link` table as well. (In our application, if a power is removed, we remove that power from the characters as well.) In order to perform a `DELETE` on multiple rows, we use the `IN` keyword, with which each id in the supplied comma-separated list of power IDs is matched against the id, and each matching row is deleted.

```
case "Delete Powers":
    if ($powers != "") {
        $powerlist = implode(',', $powers);

        $sql = "DELETE FROM char_power WHERE id IN ($powerlist)";
        $result = mysql_query($sql) or die(mysql_error());

        $sql = "DELETE FROM char_power_link
WHERE power_id IN ($powerlist)";
        $result = mysql_query($sql) or die(mysql_error());
    }
}
```

Chapter 9

When adding a power, we first check to make sure a value was passed (no need to run a query if there is nothing to add), and then attempt to insert the value into the power table. Once again, we use the `IGNORE` keyword in what follows to avoid duplication of power. We have mentioned that you really use `IGNORE` only on tables that have a primary key that is not autogenerated. There is an exception. `IGNORE` will not allow any duplicate data in any column that is designated as `UNIQUE`. In our `char_power` table, the power column is a `UNIQUE` column, so attempting to insert a duplicate value would result in an error. The `IGNORE` keyword prevents the insertion, so we don't get an error returned. If the power already exists, we simply return to the `poweredit.php` page and await further instructions.

```
case "Add Power":
    if ($newpower != '') {
        $sql = "INSERT IGNORE INTO char_power (id, power)
            VALUES (NULL, '$newpower')";
        $result = mysql_query($sql) or die(mysql_error());
    }
```

You should always have a `default:` option in your case statements. You don't need to do anything there, but it is good programming practice to include it. In this case, we are simply going to redirect the user back to the `charlist.php` page.

```
default:
    $redirect = 'charlist.php';
    break;
```

Finally, we reach the last command of `char_transact.php`. In order to use the `header()` function, no data can have been previously sent to the client. If it has, you will get an error. In our case, `char_transact.php` has no data sent to the client, so our `header()` function will work as advertised.

```
header("Location: $redirect");
```

Each case sets a destination page after running its queries. This command will now send the user to that destination.

One tremendous advantage to using a transaction page in this manner is that, because no data was sent to the client browser, once the browser gets to the destination page the history will have no memory of this page. Further, if the user refreshes his or her browser, it won't re-execute the transaction.. This makes for a very clean application.

For example, let's say a user starts on the Character List page. He or she clicks the Edit Powers link. From the Edit Powers page, the user enters a new power and clicks Add Power. The user might do this five times, adding five new powers. Each time, the PHP server submits the form to the transaction page and redirects the user back to the power page. However, if the user then clicks Back on his or her browser, the user is taken back to the Character List page, as if he or she just came from there. This is almost intuitive to the average user, and is the way applications should work.

Summary

Whew! We covered a lot of ground in this chapter. You learned about how to plan the design of your application, including database design. You learned how to normalize your data, so that it can easily be linked and manipulated. You created a brand new database for your Web site, and started building your Web site by creating tables, and creating the Web application needed to access and update those tables.

Congratulations! You just created your first, fully functioning Web application with a relational database backend. (That's going to look *so* good on your resume.)

This chapter is only the beginning, however. With the knowledge you gained here, you can create almost any application you desire. Here are some examples of what you could do:

- ❑ **Content Management (CMS):** Create a data entry systems that will allow users and administrators to alter the content of the Web site and your database without knowing any HTML.
- ❑ **Maintain a database of users visiting your site:** You can enable user authentication, e-mail your users to give them exciting news, sign them up for newsletters, and so on.
- ❑ **Create an online e-commerce site:** Create shopping carts where users can store the merchandise they will purchase. (This can be daunting—many choose to use a third-party shopping cart application.)
- ❑ **Create an online discussion forum where your users can go to discuss how wonderful your site looks!**

These are just a few ideas. In fact, we are going to show you how to do each of these things over the course of upcoming chapters. With a little imagination, you can come up with solutions to almost any problem you might face in building your site.

If any of the ideas presented in this chapter are difficult for you to grasp, that's okay. It is a lot of material, especially if you are a beginning programmer. The great thing about a book is that you can keep coming back! We will also be revisiting many of these concepts in later chapters. For example, in Chapter 15 where we teach you to build your own forum, we will go through database normalization again on a new set of databases. You will also have many more opportunities to create SQL queries, some familiar and some new.

For now, take some time to play with your new toy, the Character Database. You have the basic knowledge for creating even the most complex sites. You have the first incarnation installed on your server.

Now all you need to do is let all of your friends and family know about your cool new site. If only you knew how to send e-mails using PHP. Well, we'll handle that in Chapter 10.

