Capítulo 7

Trabajar con bases de datos y SQL

Habilidades y conceptos clave

- Aprender conceptos sobre bases de datos y el lenguaje estructurado de consultas (SQL, Structured Query Language)
- Añadir, editar, borrar y ver registros utilizando las bases de datos MySQL y SQLite
- Recuperar registros de bases de datos con PHP
- Validar y guardar datos de entrada del usuario en una base de datos con PHP
- Escribir programas portátiles sustentados por bases de datos

---

Una de las razones de la popularidad de PHP como lenguaje de creación de scripts para Web es su amplio soporte a diferentes bases de datos. Este soporte facilita que los desarrolladores Web creen sitios sustentados en bases de datos y que se hagan nuevos prototipos de aplicaciones Web de manera rápida y eficiente, sin demasiada complejidad.

PHP soporta más de quince diferentes motores de bases de datos, incluidos Microsoft SQL Server, IBM DB2, PostgreSQL, MySQL y Oracle. Hasta PHP 5, este soporte se proporcionaba mediante extensiones nativas de las bases de datos, cada una con sus propias características y funciones; sin embargo, esto dificultaba a los programadores el cambio de una base a otra. PHP 5 rectificó esta situación introduciendo una API común para el acceso a base de datos: las extensiones de objetos de datos de PHP (PDO, *PHP Data Objects*), que proporcionan una interfaz unificada para trabajar con bases de datos y ayudan a que los desarrolladores manipulen diferentes bases de datos de manera consistente.

En la versión 5.3 de PHP, las extensiones PDO han sido mejoradas, con soporte para más motores de bases de datos y mejoras considerables en la seguridad y el desempeño. Para fines de compatibilidad con versiones anteriores, se sigue dando soporte a las extensiones de bases de datos nativas. Dado que en muchas ocasiones tendrás que escoger entre una extensión nativa (que puede ser más veloz u ofrecer más características) y una PDO (que ofrece portabilidad y consistencia para diferentes motores de bases de datos), en este capítulo se abordan las dos opciones: te presenta una introducción sobre PDO y también explica el funcionamiento de las dos extensiones nativas más populares de PHP: las extensiones mejoradas de MySQL y las extensiones de SQLite.

Introducción a bases de datos y SQL

En la era de Internet, la información no se acumula ya en gabinetes; en cambio, se almacena como 1 y 0 digitales en bases de datos electrónicas, que son los “contenedores” de datos que

imponen cierta estructura en la información. Estas bases de datos electrónicas no sólo ocupan menos espacio físico que sus contrapartes de madera y metal; también contienen herramientas para ayudar a los usuarios a filtrar y recuperar información rápidamente utilizando diversos criterios. En particular, la mayoría de las bases de datos en la actualidad son *relacionales*, las cuales le permiten al usuario definir relaciones entre las tablas que conforman la base para realizar búsquedas y análisis más efectivos.

Actualmente hay disponibles muchos sistemas de administración de bases de datos; algunos son comerciales, otros gratuitos. Tal vez hayas oído mencionar algunos: Oracle, Microsoft Access, MySQL y PostgreSQL. Estos sistemas de bases de datos son aplicaciones poderosas, con muchas características, capaces de organizar y realizar búsquedas entre millones de registros a grandes velocidades; por ello son muy utilizados por empresas privadas y oficinas gubernamentales, en muchas ocasiones para llevar a cabo tareas de misión crítica.

Antes de comenzar con los vaivenes de la manipulación de registros con PHP, es esencial tener un claro entendimiento de los conceptos primordiales de las bases de datos. Si eres nuevo en el tema, las siguientes secciones te proporcionarán una base y también te permitirán comenzar a trabajar con ejercicios prácticos de SQL. Esta información será de utilidad para comprender el material más avanzado en secciones posteriores.

Comprender las bases de datos, registros y llaves primarias

Toda base de datos está compuesta por una o más *tablas*. Estas tablas, que estructuran los datos en filas y columnas, imponen una organización de datos. La figura 7-1 muestra una tabla típica.

El ejemplo contiene cifras por ventas de varias localidades, donde cada fila, también llamada *registro*, tiene información sobre diferentes lugares y años. Cada registro, a su vez, está dividido en columnas, también llamadas *campos*, y cada campo contiene un fragmento diferente de información. Esta estructura tabular facilita la búsqueda de registros, dentro de la tabla, que coincidan con ciertos *criterios*, por ejemplo: todos los lugares cuyas ventas sean mayores a $10 000, o las ventas de todos los lugares realizadas en el año 2008. Los registros

ID	*Año*	*Ubicación*	*Ventas* ($)
1	2007	Dallas	9 495
2	2007	Chicago	8 574
3	2007	Washington	12 929
4	2007	Nueva York	13 636
5	2007	Los Ángeles	8 748
6	2007	Boston	3 478
7	2008	Dallas	15 249
8	2008	Chicago	19 433
9	2008	Washington	3 738
10	2008	Nueva York	12 373
11	2008	Los Ángeles	16 162
12	2008	Boston	4 745

Figura 7-1 Tabla de ejemplo

en la tabla no tienen ningún orden en particular; pueden organizarse alfabéticamente, por año, por total de ventas, por lugar o cualquier otro criterio que selecciones para especificar su orden. Así las cosas, para facilitar la identificación de un registro en particular, se hace necesario añadir un atributo identificador único para cada registro, como un número en serie o un código de secuencia. En el ejemplo anterior, cada registro es identificado por un campo "ID registro" único; este campo es conocido como la *llave primaria* de la tabla.

TIP

Una manera fácil de comprender estos conceptos es con una analogía. Imagina que la base de datos es una biblioteca, y que cada tabla es un anaquel dentro de la biblioteca. Así, un registro sería el equivalente electrónico de un libro en el anaquel y el título del libro sería su llave primaria. Sin el título sería imposible distinguir fácilmente un libro de otro. (La única manera de hacerlo así sería abrir cada libro y revisar su contenido, proceso que consumiría mucho tiempo, ¡mismo que la llave primaria nos ahorra!)

Una vez que tienes información en una tabla, por lo general querrás utilizarla para responder ciertas preguntas, por ejemplo: ¿cuántos lugares vendieron más de $5 000 en 2008? A estas preguntas se les conoce como *consultas*, y a las preguntas respondidas por la base de datos a esas consultas se les conoce como *colecciones de resultados*. Las consultas se formulan utilizando el lenguaje estructurado de consultas (*SQL*).

Comprender relaciones y llaves externas

Ya sabes que una sola base de datos puede contener varias tablas. En las bases de datos relacionales, las tablas pueden vincularse entre sí por uno o más campos que compartan en común, llamados *llaves externas*. Éstas hacen posible la creación de relaciones uno-a-uno o uno-a-varios entre diferentes tablas, además de combinar datos de diversas tablas para crear colecciones de resultados más complejos.

Para comprenderlo mejor, examina la figura 7-2, que muestra tres tablas vinculadas.

IDGenero	*NombreGenero*
1	Horror
2	Crimen
3	Romance

IDAutor	*NombreAutor*
1	Stephen King
2	Danielle Steele
3	Dennis Lehane
4	Michael Connelly
5	Nora Roberts

IDLibro	*NombreLibro*	*IDGenero*	*IDAutor*
1	The Shining	1	1
2	Shutter Island	2	3
3	Pet Sematary	1	1
4	The Overlook	1	4
5	Sisters	3	2
6	Safe Harbour	3	2
7	Trunk Music	1	4
8	Blood Brothers	3	5

Figura 7-2 Relaciones entre tablas

La figura 7-2 muestra tres tablas, que contienen información sobre autores (tabla A), géneros (tabla G) y libros (tabla B), respectivamente. Las tablas G y B son claras: contienen una lista de géneros y nombres de autores, respectivamente, con cada registro identificado con una llave primaria única. La tabla B es un poco más compleja: cada libro de la tabla está vinculado con un género específico mediante la llave primaria perteneciente al género (tabla G) y a un autor mediante la llave primaria de autor (tabla A).

Siguiendo estas llaves hasta sus respectivas tablas fuente, es fácil identificar el autor y género de un libro específico. Por ejemplo, se puede ver que el libro "The Shining" fue escrito por "Stephen King" y pertenece al género "Horror". De manera similar, comenzando por el punto opuesto, se puede ver que el autor "Michael Connelly" ha escrito dos libros: "The Overlook" y "Trunk Music".

Relaciones como las que aparecen en la figura 7-2 son el fundamento de las bases de datos relacionales. Vincular tablas utilizando llaves externas es también más eficiente que su alternativa: crear una sola tabla que incluya hasta el nombre del perico, para almacenar toda la información, puede parecer conveniente a primera vista; pero actualizar una tabla así exige siempre una tarea manual (y propensa a errores) para localizar cada elemento de un valor específico y reemplazarlo con un nuevo valor. Dividir la información en tablas independientes y vincularlas con llaves externas asegura que una pieza de información aparece una, y sólo una vez, en la base de datos. Con esto se eliminan las redundancias, se simplifican las búsquedas (al estar localizables en un solo lugar), y se hace que la base sea más compacta y manejable.

Al proceso de afinar la base de datos definiendo y aplicando relaciones uno-a-uno y uno-a-varios entre las tablas que la integran se le conoce como *normalización de la base de datos*, y es una tarea clave que enfrentan los ingenieros informáticos cuando crean una nueva base de datos. En el proceso de normalización, el ingeniero también identifica relaciones cruzadas y dependencias incorrectas entre tablas; también optimiza la organización de los datos con el fin de que las consultas SQL se ejecuten a su máxima eficiencia. Existen *formas de normalización* disponibles que te ayudan a probar hasta qué punto está normalizada tu base de datos; estas normas proporcionan una guía útil para ayudarte a estar seguro de que el diseño de tu base tiene una estructura consistente y eficiente por igual.

Comprender las declaraciones SQL

El lenguaje estructurado de consultas, SQL, es el lenguaje estándar utilizado para comunicarse con la base de datos, añadir o cambiar registros y privilegios de usuario, y para realizar consultas. Casi todos los RDBMS comerciales utilizan en la actualidad este lenguaje, que se convirtió en estándar ANSI en 1989.

Las declaraciones SQL caen en alguna de las siguientes categorías:

- **Lenguaje de definición de datos (DLL, Data Definition Language)** Consta de declaraciones que definen la estructura y las relaciones de la base de datos y sus tablas. Por lo

general, estas declaraciones se utilizan para crear, borrar y modificar bases de datos y tablas; especificar nombres de campos y tipos; así como establecer índices.

- **Lenguaje de manipulación de datos (DML, Data Manipulation Language)** Estas declaraciones alteran o extraen datos de una base; son utilizadas para añadir y borrar registros. También se ocupan para realizar consultas, recuperar registros de una tabla que coincidan con uno o más criterios especificados por el usuario, y unir tablas utilizando los campos que comparten.
- **Lenguaje para el control de datos (DCL, Data Control Language)** Estas declaraciones se utilizan para definir acceso a diferentes niveles y privilegios de seguridad en la base de datos. Utilizarás estas declaraciones para otorgar o negar privilegios a los usuarios, asignar funciones, cambiar claves de acceso, ver permisos y crear colecciones de resultados para proteger el acceso a los datos.

Los comandos SQL imitan la lengua inglesa, lo que facilita su aprendizaje. La sintaxis también es muy intuitiva: cada declaración SQL inicia con una "palabra de acción", como DELETE (borrar), INSERT (insertar), ALTER (alterar) o DESCRIBE (describir) y termina con un punto y coma (;). Los espacios en blanco, tabuladores y retornos de carro son ignorados. A continuación aparecen algunos ejemplos de declaraciones válidas SQL:

```
CREATE DATABASE biblioteca;
SELECT película FROM películas WHERE calificación > 4;
DELETE FROM autos WHERE año_de_manufactura < 1980;
```

La tabla 7-1 muestra la sintaxis de algunas declaraciones SQL comunes, con su respectiva explicación.

Declaración SQL	Lo que hace
`CREATE DATABASE` *`nombre de la base`*	Crea una nueva base de datos
`CREATE TABLE` *`nombre de la tabla`* `(`*`campo1, campo2, ...`*`)`	Crea una nueva tabla
`INSERT INTO` *`nombre de la tabla`* `(campo1, campo2, ...) VALUES (`*`valor1, valor2, ...`*`)`	Inserta un nuevo registro en una tabla con valores específicos
`UPDATE` *`nombre de la tabla`* `SET` *`campo1=valor1, campo2=valor2, ...`* `[WHERE condición]`	Actualiza registros en una tabla con nuevos valores
`DELETE FROM` *`nombre de la tabla`* `[WHERE condición]`	Borra registros de una tabla
`SELECT` *`campo1, campo2, ...`* `FROM` *`nombre de la tabla`* `[WHERE condición]`	Recupera los registros de una tabla que coinciden con los criterios de búsqueda
`RENAME` *`nombre de la tabla`* `TO` *`nuevo nombre de la tabla`*	Cambia el nombre de una tabla
`DROP TABLE` *`nombre de la tabla`*	Borra una tabla
`DROP DATABASE` *`nombre de la base`*	Borra una base de datos

Tabla 7-1 Declaraciones SQL comunes

Pregunta al experto

P: **¿Qué tanto soporte tiene SQL?**

R: SQL es un estándar tanto ANSI como ISO, y está ampliamente respaldado por todas las bases de datos de compilación estándar SQL. Es decir, muchos proveedores de bases de datos cuentan también con "estándares" SQL extendidos con sus propias extensiones, para ofrecer a sus clientes un conjunto mejorado de características o un mejor rendimiento. Tales extensiones varían de proveedor a proveedor, y es posible que las declaraciones SQL que las utilizan no funcionen de la misma manera en todas las bases de datos. Por lo tanto, siempre es conveniente revisar la documentación de la base de datos con la que trabajes, para saber si ofrece extensiones particulares y cómo afectan las declaraciones SQL que escribas.

Prueba esto 7-1 Crear y alimentar una base de datos

Ahora que conoces los fundamentos, hagamos un ejercicio práctico que te debe familiarizar con el uso de las bases de datos y SQL. En esta sección utilizarás la línea de comando interactiva de cliente de MySQL para crear una base de datos y tablas, añadir y editar registros y generar colecciones de resultados que coincidan con varios criterios.

NOTA

A lo largo del siguiente ejercicio las palabras en negritas indican las instrucciones que debes escribir en la línea de comandos de MySQL. Las instrucciones, también llamadas "comandos", pueden escribirse en mayúsculas o minúsculas. Antes de comenzar con el ejercicio, asegúrate de haber instalado, configurado y probado la base de datos MySQL, de acuerdo con las instrucciones que aparecen en el apéndice A de este libro.

Comienza por iniciar la consola cliente de MySQL y conectarla a la base de datos con tu nombre de usuario y contraseña:

```
shell> mysql -u user -p
Contraword: ******
```

Si todo resulta bien, verás un mensaje de bienvenida y el indicador interactivo SQL, como el siguiente:

```
mysql>
```

Ahora puedes ingresar declaraciones SQL en este indicador. Las declaraciones serán transmitidas al servidor MySQL y ejecutadas en éste, y el resultado será mostrado en las líneas posteriores al indicador. Recuerda terminar cada declaración con un punto y coma.

(*continúa*)

Crear la base de datos

Como todas las tablas se almacenan en una base de datos, el primer paso consiste en crear una de éstas, utilizando la declaración `CREATE DATABASE`:

```
mysql> CREATE DATABASE musica;
Query OK, 1 row affected (0.05 sec)
```

A continuación, selecciona esta base de datos recién creada como la que se usará por omisión para todos los futuros comandos; para ello utiliza la declaración `USE`:

```
mysql> USE musica;
Database changed
```

Añadir tablas

Una vez que has inicializado tu base de datos, es hora de añadirle algunas tablas. El comando SQL para realizar esta tarea es la declaración `CREATE TABLE`, que requiere un nombre de tabla y una descripción detallada de sus campos. He aquí un ejemplo:

```
mysql> CREATE TABLE artistas (
    -> artista_id INT(4) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    -> artista_nombre VARCHAR (50) NOT NULL,
    -> artista_pais CHAR (2) NOT NULL
    -> );
Query OK, 0 rows affected (0.07 sec)
```

Esta declaración crea una tabla llamada *artistas* con tres campos: *artista_id*, *artista_nombre* y *artista_pais*. Advierte que cada nombre de campo va seguido por una *declaración de tipo*; esta declaración identifica el tipo de dato que almacenará el campo, ya sea una cadena de caracteres, numérico, temporal o booleano. MySQL soporta diferentes tipos de datos y los más importantes están resumidos en la tabla 7-2.

En el anterior ejemplo, hay otras pocas indicaciones (*modificadores*) adicionales que se establecen en la tabla:

- El modificador `NOT NULL` asegura que el campo no pueda recibir valores NULL después de cada definición de campo.
- El modificador `PRIMARY KEY` marca el campo correspondiente a la llave primaria de la tabla.
- El modificador `AUTO_INCREMENT`, que sólo es aplicable a campos numéricos, le indica a MySQL que genere automáticamente valores para este campo cada vez que se inserta un nuevo registro en la tabla, incrementando en 1 el valor previo.

Tipo de campo	Descripción
INT	Tipo numérico que puede aceptar un rango de valores de –2147483648 a 2147483647
DECIMAL	Tipo numérico que soporta valores de punto flotante y decimales
DATE	Campo de fecha con el formato AAAA-MM-DD
TIME	Campo de tiempo en formato HH:MM:SS
DATETIME	Tipo que combina fecha y hora en formato AAAA-MM-DD HH:MM:SS
YEAR	Campo específico para años que acepta un rango de 1901 a 2155; en formatos AAAA o AA
TIMESTAMP	Tipo sello cronológico en formato AAAAMMDDHHMMSS
CHAR	Tipo cadena de caracteres con un tamaño máximo de 255 caracteres y longitud fija
VARCHAR	Tipo cadena de caracteres con un tamaño máximo de 255 caracteres y longitud variable
TEXT	Tipo cadena de caracteres con un tamaño máximo de 65535 caracteres
BLOB	Tipo binario para datos variables
ENUM	Tipo cadena de caracteres que acepta un valor de una lista de posibles valores definida con anterioridad
SET	Tipo cadena de caracteres que acepta cero o más valores de un conjunto de posibles valores definido con anterioridad

Tabla 7-2 Tipos de datos MySQL

Ahora sigamos adelante para crear otras dos tablas utilizando estas declaraciones SQL:

```
mysql> CREATE TABLE ratings (
    -> rating_id INT(2) NOT NULL PRIMARY KEY,
    -> rating_nombre VARCHAR (50) NOT NULL
    -> );
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE canciones (
    -> cancion_id INT(4) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    -> cancion_titulo VARCHAR(100) NOT NULL,
    -> ex_cancion_artista INT(4) NOT NULL,
    -> ex_cancion_rating INT(2) NOT NULL
    -> );
Query OK, 0 rows affected (0.05 sec)
```

(*continúa*)

Añadir registros

Añadir registros a la tabla es tan sencillo como invocar la declaración INSERT con los valores apropiados. He aquí un ejemplo, que añade un registro a la tabla *artistas* especificando valores para los campos *artista_id* y *artista_nombre*:

```
mysql> INSERT INTO artistas (artista_id, artista_nombre, artista_pais)
    -> VALUES ('1', 'Aerosmith', 'US');
Query OK, 1 row affected (0.00 sec)
```

Recordarás, de la sección anterior, que el campo *artista_id* fue marcado con el modificador AUTO_INCREMENT. Ésta es una extensión MySQL sobre el SQL estándar, que indica a MySQL que asigne un valor automáticamente a este campo en caso de quedar sin especificación. Para verlo en acción trata de añadir otro registro utilizando la siguiente declaración:

```
mysql> INSERT INTO artistas (artista_nombre, artista_pais)
    -> VALUES ('Abba', 'SE');
Query OK, 1 row affected (0.00 sec)
```

De manera similar, añade algunos registros a la tabla *ratings*:

```
mysql> INSERT INTO ratings (rating_id, rating_nombre) VALUES (4, 'Bueno');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO ratings (rating_id, rating_nombre) VALUES (5, 'Excelente');
Query OK, 1 row affected (0.00 sec)
```

Y algunos a la tabla *canciones*:

```
mysql> INSERT INTO canciones (cancion_titulo, ex_cancion_artista,
    ex_cancion_rating) -> VALUES ('Janie\'s Got a Gun', 1, 4);
Query OK, 1 row affected (0.04 sec)
mysql> INSERT INTO canciones (cancion_titulo, ex_cancion_artista,
    ex_cancion_rating) -> VALUES ('Crazy', 1, 5);
Query OK, 1 row affected (0.00 sec)
```

Advierte que los registros en la tabla *canciones* están vinculados a los registros de la tabla *artistas* y *ratings* por llaves externas. En la siguiente sección verás en acción este tipo de relaciones por llave externa.

NOTA

El archivo de código de este libro tiene una lista completa de declaraciones SQL INSERT para llenar las tres tablas utilizadas en este ejercicio. Ejecuta estas declaraciones y termina de construir las tablas antes de pasar a la siguiente sección.

Ejecutar consultas

Una vez que los datos están en la base, es hora de hacer algo con ellos. SQL te permite buscar registros que coinciden con ciertos criterios específicos utilizando la declaración `SELECT`. He aquí un ejemplo que regresa todos los registros de la tabla *artistas*:

```
mysql> SELECT artista_id, artista_nombre FROM artistas;
+------------+----------------+
| artista_id | artista_nombre |
+------------+----------------+
|          1 | Aerosmith      |
|          2 | Abba           |
|          3 | Timbaland      |
|          4 | Take That      |
|          5 | Girls Aloud    |
|          6 | Cubanismo      |
+------------+----------------+
6 rows in set (0.00 sec)
```

En casi todos los casos querrás añadir filtros a tu consulta, para reducir el tamaño de la colección de resultados y asegurar que contiene sólo los registros que coinciden con ciertos criterios. Esto se hace añadiendo la cláusula `WHERE` a la declaración `SELECT` junto con una o más expresiones condicionales. He aquí un ejemplo que muestra sólo los artistas de Estados Unidos:

```
mysql> SELECT artista_id, artista_nombre FROM artistas
    -> WHERE artista_pais = 'US';
+------------+----------------+
| artista_id | artista_nombre |
+------------+----------------+
|          1 | Aerosmith      |
|          3 | Timbaland      |
+------------+----------------+
2 rows in set (0.00 sec)
```

Todos los operadores de comparación estándar con los que ya estás familiarizado por PHP tienen soporte en SQL. El ejemplo anterior utiliza el operador de igualdad (=); el siguiente ejemplo muestra el funcionamiento del operador "mayor que o igual a" (>=), que regresa una lista con las canciones con rating 4 o superior:

```
mysql> SELECT cancion_titulo, ex_cancion_rating FROM canciones
    -> WHERE ex_cancion_rating >= 4;
+-------------------+-------------------+
| canción_título    | ex_canción_rating |
+-------------------+-------------------+
| Janie's Got A Gun |                 4 |
| Crazy             |                 5 |
| En Las Delicious  |                 5 |
```

(*continúa*)

```
| Pray               |                 4 |
| Apologize          |                 4 |
| SOS                |                 4 |
| Dancing Queen      |                 4 |
+--------------------+-------------------+
7 rows in set (0.00 sec)
```

Puedes combinar expresiones condicionales utilizando los operadores lógicos `AND`, `OR` y `NOT` (tal y como se hace en una declaración condicional regular PHP). He aquí un ejemplo, que presenta una lista con los artistas de Estados Unidos y el Reino Unido:

```
mysql> SELECT artista_nombre, artista_pais FROM artistas
    -> WHERE artista_pais = 'US'
    -> OR artista_pais = 'UK';
+----------------+--------------+
| artista_nombre | artista_país |
+----------------+--------------+
| Aerosmith      | US           |
| Timbaland      | US           |
| Take That      | UK           |
| Girls Aloud    | UK           |
+----------------+--------------+
4 rows in set (0.02 sec)
```

Ordenar y limitar colecciones de resultados

Si quieres ver los datos de una tabla ordenados por un campo específico, SQL cuenta con la cláusula `ORDER BY`. Te permite definir el nombre del campo sobre el que desees basar el orden del resultado y su dirección (ascendente o descendente).

Por ejemplo, para ver la lista de las canciones en orden alfabético, utiliza la siguiente declaración SQL:

```
mysql> SELECT cancion_titulo FROM canciones
    -> ORDER BY cancion_titulo;
+--------------------------+
| canción_título           |
+--------------------------+
| Another Crack In My Heart |
| Apologize                |
| Babe                     |
| Crazy                    |
| Dancing Queen            |
| En Las Delicious         |
| Gimme Gimme Gimme        |
| Janie's Got A Gun        |
| Pray                     |
| SOS                      |
```

```
| Sure                       |
| Voulez Vous                |
+----------------------------+
12 rows in set (0.04 sec)
```

Para invertir el orden de la lista, añade el modificador `DESC`, como se muestra a continuación:

```
mysql> SELECT cancion_titulo FROM canciones
    -> ORDER BY cancion_titulo DESC;
+----------------------------+
| canción_título             |
+----------------------------+
| Voulez Vous                |
| Sure                       |
| SOS                        |
| Pray                       |
| Janie's Got A Gun          |
| Gimme Gimme Gimme          |
| En Las Delicious           |
| Dancing Queen              |
| Crazy                      |
| Babe                       |
| Apologize                  |
| Another Crack In My Heart  |
+----------------------------+
12 rows in set (0.00 sec)
```

SQL también te permite limitar la cantidad de registros que aparecen en la colección de resultados con la palabra clave `LIMIT`, que acepta dos parámetros: la posición del registro para comenzar (a partir de 0) y la cantidad de registros que aparecerán. Por ejemplo, para mostrar las filas 4-9 (inclusivas) en una colección de resultados, utiliza la siguiente declaración:

```
mysql> SELECT cancion_titulo FROM canciones
    -> ORDER BY cancion_titulo
    -> LIMIT 3,6;
+--------------------+
| canción_título     |
+--------------------+
| Crazy              |
| Dancing Queen      |
| En Las Delicious   |
| Gimme Gimme Gimme  |
| Janie's Got A Gun  |
| Pray               |
+--------------------+
5 rows in set (0.00 sec)
```

(*continúa*)

Utilizar comodines

La declaración `SELECT` de SQL también da soporte a la cláusula `LIKE`, que puede utilizarse para realizar búsquedas dentro de campos de texto con el uso de comodines. Hay dos tipos de comodines aceptados en la cláusula `LIKE`: el signo de porcentaje (`%`), que es utilizado para representar cero o más apariciones de cierto carácter y el guión bajo (_), cuyo uso significa exactamente una aparición de cierto carácter.

El siguiente ejemplo muestra la cláusula `LIKE` en acción, que busca títulos de canciones que contengan el carácter 'g':

```
mysql> SELECT cancion_id, cancion_titulo FROM canciones
    -> WHERE cancion_titulo LIKE '%g%';

+------------+-------------------+
| canción_id | canción_título    |
+------------+-------------------+
|          1 | Janie's Got A Gun |
|          7 | Apologize         |
|          8 | Gimme Gimme Gimme |
|         10 | Dancing Queen     |
+------------+-------------------+
4 rows in set (0.00 sec)
```

Fusionar tablas

Hasta ahora, todas las consultas que has visto se concentran en una sola tabla. Pero SQL también te permite realizar búsquedas entre dos o más tablas al mismo tiempo y combinar el resultado en una sola colección de resultados. Esta tarea lleva el nombre técnico de *fusión*, pues "fusiona" diferentes tablas que comparten campos entre sí (las llaves externas) para crear nuevas vistas de los datos.

TIP

Cuando fusiones tablas, usa un prefijo para cada nombre de campo con el nombre de la tabla a la que pertenece, con el fin de evitar confusiones en caso de que varios campos en diferentes tablas tengan el mismo nombre.

He aquí un ejemplo de fusión entre las tablas *canciones* y *artistas* utilizando el campo común *artista_id* (la cláusula `WHERE` es utilizada para transformar los campos comunes entre sí):

```
mysql> SELECT cancion_id, cancion_titulo, artista_nombre FROM canciones,
artistas
    -> WHERE canciones.ex_cancion_artista = artistas.artista_id;
```

```
+------------+--------------------------+----------------+
| canción_id | canción_título           | artista_nombre |
+------------+--------------------------+----------------+
|          1 | Janie's Got A Gun        | Aerosmith      |
|          2 | Crazy                    | Aerosmith      |
|          8 | Gimme Gimme Gimme        | Abba           |
|          9 | SOS                      | Abba           |
|         10 | Dancing Queen            | Abba           |
|         11 | Voulez Vous              | Abba           |
|          7 | Apologize                | Timbaland      |
|          4 | Sure                     | Take That      |
|          5 | Pray                     | Take That      |
|          6 | Another Crack In My Heart | Take That      |
|         12 | Babe                     | Take That      |
|          3 | En Las Delicious         | Cubanismo      |
+------------+--------------------------+----------------+
12 rows in set (0.00 sec)
```

Y a continuación tenemos un ejemplo que fusiona las tres tablas y luego filtra la colección de resultados aún más para incluir sólo las canciones con rating 4 o superior de artistas que no sean de Estados Unidos:

```
mysql> SELECT cancion_titulo, artista_nombre, rating_nombre
    -> FROM canciones, artistas, ratings
    -> WHERE canciones.ex_cancion_artista = artistas.artista_id
    -> AND canciones.ex_cancion_rating = ratings.rating_id
    -> AND ratings.rating_id >= 4
    -> AND artistas.artista_pais != 'US';
+-----------------+----------------+---------------+
| canción_título  | artista_nombre | rating_nombre |
+-----------------+----------------+---------------+
| En Las Delicious | Cubanismo      | Excelente     |
| Pray            | Take That      | Buena         |
| SOS             | Abba           | Buena         |
| Dancing Queen   | Abba           | Buena         |
+-----------------+----------------+---------------+
4 rows in set (0.02 sec)
```

Modificar y eliminar registros

Así como insertas registros en una tabla (con el comando `INSERT`), también es posible borrarlos con la declaración `DELETE`. Por lo general, seleccionarás un subgrupo específico de filas para ser borradas añadiendo la cláusula `WHERE` a la declaración `DELETE`, como se muestra en el siguiente ejemplo, que elimina todas las canciones con un rating igual a o menor que 3:

```
mysql> DELETE FROM canciones
    -> WHERE ex_cancion_rating <= 3;
Query OK, 5 rows affected (0.02 sec)
```

(*continúa*)

También existe una declaración UPDATE, que puede utilizarse para cambiar el contenido de un registro; esta declaración también acepta la cláusula WHERE, de manera que puedes aplicar los cambios sólo en los registros que coinciden con cierto criterio. Examina el siguiente ejemplo, que cambia el rating 'Excelente' por 'Fantástico':

```
mysql> UPDATE ratings SET rating_nombre = 'Fantástico'
    -> WHERE rating_nombre = 'Excelente';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Puedes modificar varios campos separándolos con comas. He aquí un ejemplo que actualiza el registro de una canción en particular en la tabla *canciones*, modificando tanto el título como el rating:

```
mysql> UPDATE ratings SET cancion_titulo = 'Waterloo',
    -> ex_cancion_rating = 5
    -> WHERE cancion_id = 9;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Pregunta al experto

P: Estoy utilizando la base de datos ____ y no soporta el tipo de campo ____. ¿Qué hago ahora?

R: Diferentes bases de datos tienen distinta sintaxis para tipos de datos de los campos. Por ejemplo, MySQL llama a los campos enteros INT, mientras que SQLite llama a los mismos campos INTEGER. De cualquier manera, casi todas las bases de datos soportan (por lo menos) tipos de datos para valores enteros, de punto flotante, cadenas de caracteres, binarios y NULL. Lo único que necesitas es revisar la documentación de tu base de datos y encontrar la sintaxis para utilizar correctamente los tipos de datos en tus declaraciones SQL.

P: ¿Por qué necesito incluir todos los campos requeridos en la declaración SELECT? ¿No puedo utilizar el comodín * para recuperar todos los campos disponibles?

R: SQL sí soporta el asterisco (*) como carácter comodín para representar "todos los campos de la tabla", pero es preferible siempre designar explícitamente los campos que quieras ver en la colección de resultados. Esto permite que la aplicación sobreviva a los cambios estructurales en las tablas (o la tabla), además de que es más eficiente en el uso de memoria porque la colección de resultados no contendrá datos irrelevantes o no deseados.

Utilizar la extensión MySQLi de PHP

Como ya se explicó, PHP permite que los desarrolladores interactúen con la base de datos de dos maneras: utilizando extensiones personalizadas específicas de la base, o la extensión neutral (PDO). Mientras que estas últimas son más portátiles, muchos desarrolladores encuentran preferible utilizar las extensiones nativas de la base de datos con la que trabajan, sobre todo cuando ofrecen mejor rendimiento o más características que la versión PDO.

De los diferentes motores de base de datos soportados por PHP, el más popular es MySQL. No resulta difícil entender el porqué: tanto PHP como MySQL son proyectos de código libre, y al utilizarlos juntos, los desarrolladores obtienen beneficios de los grandes ahorros en costos de licencias en comparación con las opciones comerciales. Históricamente, además, PHP ha ofrecido soporte extraoficial para MySQL desde la versión 3, y tiene sentido aprovechar el tremendo esfuerzo que han realizado los desarrolladores de PHP y MySQL para asegurar que los dos paquetes trabajen juntos sin problemas.

Además de dar soporte a MySQL mediante las extensiones de objetos de datos de PHP (que se abordarán en la siguiente sección), PHP también incluye una extensión MySQL personalizada que lleva el nombre de MySQL Mejorado (MySQLi Improved). Esta extensión brinda beneficios de velocidad y de características superiores a la versión PDO y es una buena opción para desarrollar proyectos específicos con MySQL. Las siguientes secciones abordan esta extensión con gran detalle.

Recuperar datos

En la sección anterior creaste una base de datos y utilizaste una consulta `SELECT` para recuperar una colección de resultados de ella. Ahora harás lo mismo utilizando PHP, como en el siguiente script:

```
<?php
// intenta conectarse con la base de datos
$mysqli = new mysqli("localhost", "user" "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// intenta ejecutar consulta
// itera sobre colección de resultados
// muestra cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $mysqli ->query($sql)) {
  if ($result->num_rows > 0) {
    while($row = $result->fetch_array()) {
      echo $row[0] . ":" . $row[1] . "\n";
    }
```

```
    $result->close();
  } else {
    echo "No se encontró ningún registro que coincida con su búsqueda.";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql. " . $mysqli->error;
}

// cierra conexión
$mysqli->close();
?>
```

La figura 7-3 muestra cómo se debe ver el resultado del script.

Como puedes ver, utilizar PHP para obtener datos de una base requiere varios pasos, que se describen a continuación:

1. Con el fin de establecer comunicación con el servidor que contiene la base de datos MySQL, primero necesitas abrir una conexión con el mismo. Toda la comunicación entre PHP y el servidor de base de datos se realiza a través de esta conexión.

Para inicializar esta conexión, inicializa un objeto de la clase MySQLi y pasa cuatro argumentos al constructor del objeto: el nombre del servidor anfitrión de MySQL al que intentas conectarte, un nombre y una contraeña válidos para obtener el acceso necesario, y el nombre de la base de datos que quieres utilizar.

Dando por hecho que la conexión se estableció con éxito, este objeto representa la conexión con la base de datos para todas las futuras operaciones y expone los métodos para realizar consultas, búsquedas y para procesar las colecciones de resultados. Si el intento de conexión fracasó, el objeto será falso; entonces es posible obtener un mensaje de error que explique las razones de la falla; esto se lleva a cabo invocando la función `mysqli_connect_error()`.

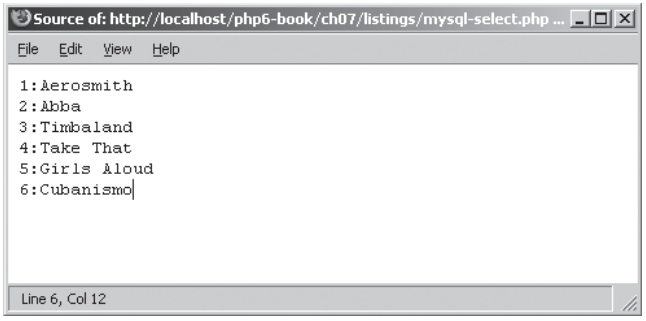

Figura 7-3 Registros recuperados de una base de datos MySQL con PHP

TIP

Si los dos servidores, el de base de datos y el Web, se ejecutan en la misma computadora, puedes utilizar `localhost` como nombre de servidor.

2. El siguiente paso consiste en crear y ejecutar la consulta SQL. Esto se realiza invocando el método `query()` para el objeto de MySQLi y transmitiéndole la consulta que se va a ejecutar. Si la consulta no tuvo éxito, el método regresa un valor booleano falso, y un mensaje de error que explica la causa de la falla es almacenado en la propiedad `'error'` del objeto MySQLi.
3. Si, por otra parte, la consulta se ejecuta con éxito y regresa uno o más registros, el valor de retorno del método `query()` se convierte en otro objeto, este último una instancia de la clase MySQLi_Result. Este objeto representa la colección de resultados regresada por la consulta, y expone varios métodos para procesar los registros individuales en la colección de resultados.

   Uno de esos métodos es `fetch_array()`. Cada vez que se invoca, regresa el siguiente registro de la colección de resultados como una matriz. Esto hace que el método `fetch_array()` se acople bien con el uso de los bucles `while` y `for`. El contador del bucle determina cuántas veces debe ejecutarse; este resultado se obtiene de la propiedad `'num_rows'` del objeto MySQLi_Result, que almacena el número de filas regresadas por la consulta. Los campos individuales del registro pueden accesarse como elementos de una matriz, utilizando ya sea el índice del campo o su nombre.

PRECAUCIÓN

La propiedad `'num_rows'` sólo tiene significado cuando se utiliza con consultas que regresan datos, como `SELECT`; no debe utilizarse con los consultas `INSERT`, `UPDATE` o `DELETE`.

4. Cada colección de resultados regresada después de una consulta ocupa cierta cantidad de memoria. Así, una vez que el resultado ha sido procesado, es buena idea destruir el objeto MySQLi_Result, para liberar la memoria que utiliza; esto se realiza con el método `close()`. Y una vez que has terminado de trabajar con la base de datos, también es buena idea destruir el objeto principal MySQLi de manera similar, invocando el método `close()`.

Regresar registros como matrices y objetos

El ejemplo anterior mostró un método para procesar la colección de resultados: el método `fetch_array()` del objeto MySQLi_Result. Este método regresa cada registro de la colección de resultados como una matriz que contiene llaves indexadas tanto numéricamente como por cadenas de caracteres; esto ofrece a los desarrolladores la conveniencia de hacer referencia a campos individuales de cada registro ya sea por índice o por nombre de campo.

El ejemplo anterior mostró cómo recuperar campos individuales utilizando el número de índice. El siguiente ejemplo, equivalente al anterior, realiza la misma tarea utilizando nombres de campo:

```
<?php
// intenta conectarse con la base de datos
$mysqli = new mysqli("localhost", "usuario", "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// intenta ejecutar consulta
// reitera sobre colección de resultados
// muestra cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $mysqli ->query($sql)) {
  if ($result->num_rows > 0) {
    while($row = $result->fetch_array()) {
      echo $row['artista_id'] . ":" . $row['artista_nombre'] . "\n";
    }
    $result->close();
  } else {
    echo "No se encontró ningún registro que coincida con su búsqueda.";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql. " . $mysqli->error;
}

// cierra conexión
$mysqli->close();
?>
```

Sin embargo, existe un tercer método para recuperar registros: como objetos, utilizando el método `fetch_object()`. Aquí, cada registro se representa como un objeto y los campos de un registro se representan como propiedades del objeto. Después, los campos individuales pueden ser accesados utilizando la notación estándar `$objeto->propiedad`. El siguiente ejemplo ilustra este procedimiento:

```
<?php
// intenta conectarse con la base de datos
$mysqli = new mysqli("localhost", "usuario", "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// intenta ejecutar consulta
// reitera sobre colección de resultados
```

```
// muestra cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $mysqli->query($sql)) {
  if ($result->num_rows > 0) {
    while($row = $result->fetch_object()) {
      echo $row->artista_id . ":" . $row->artista_nombre . "\n";
    }
    $result->close();
  } else {
    echo "No se encontró ningún registro que coincida con su búsqueda.";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql. " . $mysqli->error;
}

// cierra conexión
$mysqli->close();
?>
```

Añadir y modificar datos

Es igual de sencillo ejecutar un consulta que cambie los datos en la base, ya sea insertar (`INSERT`), actualizar (`UPDATE`) o borrar (`DELETE`). El siguiente ejemplo lo muestra, añadiendo un nuevo registro a la tabla *artistas*:

```
<?php
// intenta conectarse con la base de datos
$mysqli = new mysqli("localhost", "usuario", "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// intenta ejecutar consulta
// añade un nuevo registro
// datos de salida: " Nuevo artista con id: 7 ha sido añadido. "
$sql = "INSERT INTO artistas (artista_nombre, artista_pais) VALUES
('Kylie Minogue', 'AU')";
if ($mysqli->query($sql)=== true) {
  echo 'Nuevo artista con id: ' . $mysqli->insert_id . 'ha sido
  añadido.';
} else {
  echo "ERROR: No fue posible ejecutar $sql. " . $mysqli->error;
}

// cierra conexión
$mysqli->close();
?>
```

Como verás, esto no es muy diferente de la necesidad de aplicar una consulta `SELECT` al programar. De hecho, es un poco más sencillo porque no hay una colección de datos para procesar; todo lo que se necesita es probar el valor regresado del método `query()` y verificar si la consulta se ejecutó correctamente o no. Advierte también el uso de la nueva propiedad '`insert_id`', que regresa el ID generado por la última consulta `INSERT` (sólo es útil si la tabla en la cual se aplicó `INSERT` contiene un campo que se incrementa automáticamente).

¿Qué hay de incrementar un registro existente? Lo único que necesitas es cambiar la línea de comando SQL:

```
<?php
// intenta conectarse con la base de datos
$mysqli = new mysqli("localhost", "usuario", "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// intenta ejecutar consulta
// añade un nuevo registro
// datos de salida: "1 fila(s) actualizadas."
$sql = "UPDATE artistas SET artista_nombre = 'Eminem', artista_pais =
'US' WHERE artista_id = 7;
if ($mysqli->query($sql)=== true) {
  echo $mysqli->affected_rows . ' fila(s) actualizadas.';
} else {
  echo "ERROR: No fue posible ejecutar: $sql. " . $mysqli->error;
}

// cierra conexión
$mysqli->close();
?>
```

Cuando ejecutas `UPDATE` o `DELETE`, el número de filas afectadas por la declaración se almacenarán en la propiedad '`affected_rows`' del objeto MySQLi. El ejemplo siguiente muestra el uso de esta propiedad.

Utilizar declaraciones preparadas

En caso de que necesites ejecutar un consulta particular varias veces con diferentes valores (por ejemplo, una serie de declaraciones `INSERT`), el servidor MySQL da soporte a *declaraciones preparadas*, que ofrecen medios más eficientes para completar esta tarea en lugar de invocar repetidamente el método `$mysqli->query()`.

En esencia, una declaración preparada es una consulta SQL modelo que contiene variables prealmacenadas para los valores que serán insertados o modificados. Esta declaración es almacenada en el servidor de base de datos y la invoca todas las veces que sea necesario; las variables almacenadas previamente se reemplazan con los nuevos valores cada vez que se ejecutan. Dado que la declaración está almacenada en el servidor de base de datos, una decla-

ración preparada suele ser más rápida para realizar operaciones por lote que implican ejecutar la misma consulta SQL una y otra vez con diferentes valores.

Pregunta al experto

P: ¿Por qué las declaraciones preparadas ofrecen mejor rendimiento?

R: En condiciones normales, cada vez que se ejecuta una declaración SQL sobre el servidor de base de datos, éste debe segmentar el código SQL y verificar su sintaxis y estructura antes de permitir su ejecución. Con una declaración preparada, la declaración SQL está almacenada temporalmente en el servidor de base de datos y, por lo mismo, sólo necesita segmentarse y validarse una sola vez. Más aún, cada vez que se ejecuta la declaración, sólo los valores que cambian necesitan transmitirse al servidor, en lugar de enviar la declaración completa.

Para ver una declaración preparada en acción, examina el siguiente script, que inserta varias canciones en la base de datos utilizando precisamente una declaración preparada con la extensión MySQLi de PHP:

```
<?php
// define valores a ser insertados
$canciones = array(
  array('Patience', 4, 3),
  array('Beautiful World', 4, 4),
  array('Shine', 4, 4),
  array('Hold On', 4, 3),
);

// intenta establecer la conexión con la base de datos
$mysqli = new mysqli("localhost", "usuario", "contra", "musica");
if ($mysqli === false) {
  die ("ERROR: No se estableció la conexión. " . mysqli_connect_error());
}

// prepara el consulta modelo
// lo ejecuta varias veces
$sql = "INSERT INTO canciones (cancion_titulo, ex_cancion_artista, ex_
cancion_rating) VALUES (?, ?, ?)";
if ($stmt = $mysqli->prepare($sql)) {
  foreach ($canciones as $s) {
    $stmt->bind_param('sii', $s[0], $s[1], $s[2]);
    if ($stmt->execute()) {
     echo "Nueva canción con id: " . mysqli->insert_id . "ha sido añadida.\n";
    } else {
     echo "ERROR: No fue posible ejecutar consulta: $sql. " . $mysqli->error;
     }
  }
```

```
} else {
  echo "ERROR: No fue posible preparar consulta: $sql. " . $mysqli
->error;
}

// cierra conexión
$mysqli->close();
?>
```

Como se aprecia en este ejemplo, el uso de una declaración preparada implica un proceso diferente al que has visto en ejemplos previos.

Para utilizar una declaración preparada, primero es necesario definir la cadena de la consulta que será ejecutada varias veces. Esta cadena de consulta por lo regular contendrá uno o varios contenedores, representados por los signos de interrogación (`?`). Estos contenedores serán sustituidos por los nuevos valores cada vez que se ejecute la declaración. Entonces se transmite la cadena de consulta con sus contenedores al método `prepare()` del objeto MySQLi, que verifica el comando SQL en busca de errores y regresa un objeto MySQLi_Stmt que representa la declaración preparada.

Ahora, ejecutar la declaración preparada es cuestión de realizar dos acciones, por lo general con un bucle:

1. *Unir los valores a la declaración preparada*. Los valores que van a ser interpolados en la declaración necesitan *unirse* a sus contenedores con el método `bind_param()` del objeto MySQLi_Stmt. El primer argumento para este método debe ser una cadena de secuencia ordenada que indique los tipos de datos de los valores que serán interpolados (`s` para una cadena, `i` para un entero, `d` para un número de doble precisión); este argumento es seguido por los nuevos valores. En el ejemplo anterior, la cadena '`sii`' indica que los valores que se interpolarán en la declaración preparada serán, en secuencia, un tipo cadena de caracteres (el título de la canción), un tipo entero (la llave externa del artista) y otro tipo entero (la llave externa para el rating).
2. *Ejecutar la declaración preparada*. Una vez que se han unido los valores a sus contenedores, se ejecuta la declaración preparada invocando el método `execute()` del objeto MySQLi_Stmt. Este método reemplaza los contenedores en la declaración preparada con los nuevos valores y se ejecuta en el servidor.

De esta manera, el uso de una declaración preparada ofrece beneficios de rendimiento cuando necesitas ejecutar la misma declaración varias veces, y sólo los valores cambian en cada ejecución del comando. Las bases de datos más populares, incluyendo MySQL, PostgreSQL, Oracle e InterBase, soportan declaraciones preparadas, y esta característica debe ser utilizada cuando esté disponible para realizar tus operaciones SQL por lote de manera más eficiente.

Manejo de errores

Si tu código de base de datos no funciona como esperabas, no te preocupes; la extensión MySQLi contiene una gran cantidad de funciones que te pueden decir la causa. Ya las has visto en acción en diferentes lugares de los ejemplos anteriores, pero a continuación aparece una lista completa:

- La propiedad '`error`' del objeto MySQLi guarda el último mensaje de error generado por el servidor de base de datos.
- La propiedad '`errno`' del objeto MySQLi guarda el último error de código regresado por el servidor de base de datos.
- La función `mysqli_connect_error()` regresa el mensaje de error generado por el último intento (fallido) de conexión.
- La función `mysqli_connect_errno()` regresa el error de código generado por el último intento (fallido) de conexión.

TIP

Para hacer tu aplicación más robusta contra errores, y para localizarlos y resolverlos con mayor facilidad, es una buena idea utilizar estas funciones para manejo de errores con toda libertad dentro de tu código.

Prueba esto 7-2 Añadir empleados a una base de datos

Ahora que ya conoces el uso básico de las extensiones MySQLi de PHP, practicarás lo que aprendiste en la sección anterior en una aplicación "real". La aplicación es un formulario Web que permite al usuario insertar nombres de empleados y sus puestos en una base de datos para empleados basada en MySQL. PHP valida y limpia los valores ingresados en el formulario; después se transforman en registros de la base con el uso de la extensión MySQLi de PHP.

Para empezar a construir esta aplicación, primero ejecuta el programa cliente de línea de comando MySQL y crea una base de datos vacía.

```
mysql> CREATE DATABASE empleados;
Query OK, 1 row affected (0.20 sec)
```

Después crea una tabla que contenga los registros de los empleados:

```
mysql> USE empleados;
Database changed
mysql> CREATE TABLE empleados (
```

(*continúa*)

```
    -> id INT (4) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> nombre VARCHAR(255) NOT NULL,
    -> puesto VARCHAR(255) NOT NULL
    -> );
Query OK, 0 row affected (0.20 sec)
```

¿Todo listo? Lo siguiente es construir un formulario Web que acepte los datos del empleado (en este caso, su nombre y puesto). Al momento de enviarlo, el procesador del formulario verificará los datos de entrada y, de ser válidos, generará una consulta `INSERT` para insertarlos en la tabla creada en el paso anterior.

He aquí el script (*empleados.php*):

```
<!DOCTYPE html PUBLIC "-// W3C//DTD XHTML 1.0 Transitional//EN"
  "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-2: Añadir empleados a una base de datos</title>
    <style type="text/css">
    div#message{
      text-align:center;
      margin-left:auto;
      margin-right:auto;
      width:40%;
      border: solid 2px green
    }
    </style>
  </head>
  <body>
    <h2>Proyecto 7-2: Añadir empleados a una base de datos</h2>
    <h3>Añade un Nuevo Empleado</h3>
<?php
    // si el formulario ha sido enviado
    // procesa los datos del formulario
    if (isset($_POST['submit'])) {
      // intenta conectarse con la base de datos MySQL
      $mysqli = new mysqli("localhost", "usuario" "contra", "empleados");
      if ($mysqli === false) {
        die("ERROR: No fue posible conectarse con la base de datos. " .
mysqli_connect_error());
      }

      // abre bloque de mensaje
      echo '<div id="message">';

      // recupera y verifica los valores de entrada
      $inputError = false;
      if (empty($_POST['emp_nombre'])) {
        echo 'ERROR: Por favor ingrese un nombre de empleado válido';
```

```
        $inputError = true;
      } else {
        $nombre = $mysqli->escape_string($_POST['emp_nombre']);
      }

      if ($inputError != true && empty($_POST['emp_puesto'])) {
        echo 'ERROR: Por favor ingrese un puesto válido';
        $inputError = true;
      } else {
        $puesto = $mysqli->escape_string($_POST['emp_puesto']);
      }

      // añade valores a la base de datos utilizando el consulta INSERT
      if ($inputError != true) {
      $sql = "INSERT INTO empleados (nombre, puesto)
                VALUES ('$nombre', '$puesto')";
        if ($mysqli->query($sql) === true) {
          echo 'Nuevo registro de empleado añadido con ID: ' . $mysqli
          ->insert_id;
        } else {
          echo "ERROR: No fue posible ejecutar el consulta: $sql. " .
          $mysqli->error;
        }
      }

      // cierra bloque de mensaje
      echo '</div>';

      // cierra conexión
      $mysqli->close();
    }
?>
    </div>

    <form action="empleados.php" method="POST">
      Nombre del empleado: <br />
      <input type="text" name="emp_nombre" size="40" />
      <p />
      Puesto del empleado: <br />
      <input type="text" name="emp_puesto" size="40" />
      <p />
      <input type="sumbit" name="submit" value="Enviar" />
    </form>

  </body>
</html>
```

Cuando se invoca este script, genera un formulario Web con campos para el nombre y puesto de los empleados. La figura 7-4 muestra cómo luce el formulario inicial.

(*continúa*)

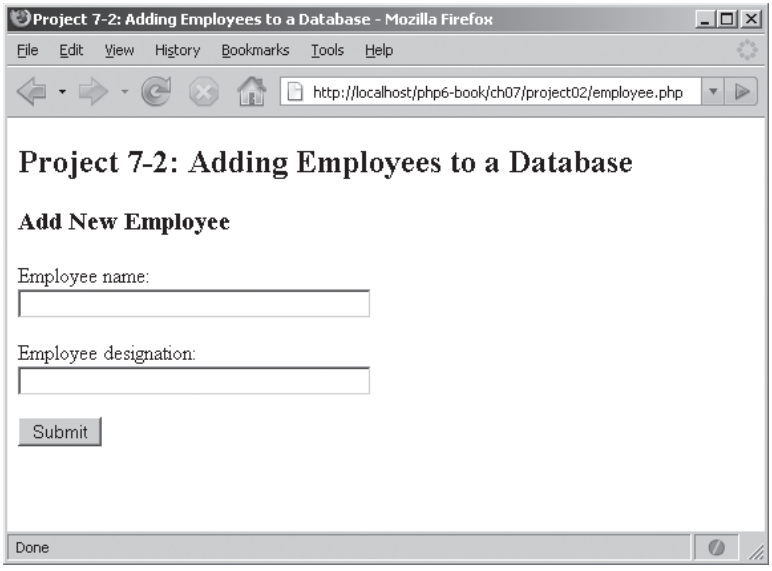

Figura 7-4 Formulario Web para insertar nuevos empleados

Cuando se envía el formulario, el script procede a inicializar un nuevo objeto MySQLi y abre la conexión para el servidor de bases de datos MySQL. Luego verifica los campos del formulario para asegurarse de que contienen valores de entrada válidos. De ser así, cada uno de estos valores es "limpiado" utilizando el método `escape_string()` del objeto MySQLi, que automáticamente limpia los caracteres especiales en los datos de entrada como preludio para insertarlos en la base de datos y luego interpola en la consulta INSERT, que se ejecuta con el método `query()` del objeto para guardar los valores en la base de datos. Los errores, si existen, se muestran utilizando la propiedad error del objeto MySQLi.

La figura 7-5 muestra los datos de salida cuando se agrega un registro correctamente y la figura 7-6 muestra los datos de salida cuando falla la validación en un campo de entrada.

Eso fue muy fácil. Ahora, ¿qué tal si rehacemos el script para que, además de permitir la inserción de nuevos empleados a la base de datos, también muestre los registros existentes en la misma? No es muy difícil; se añade otra invocación al método `query()`, esta vez con una consulta `SELECT`, y se procesa la colección de resultados utilizando un bucle.

He aquí el código modificado:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-2: Añadir empleados a una base de datos</title>
    <style type="text/css">
    div#message {
```

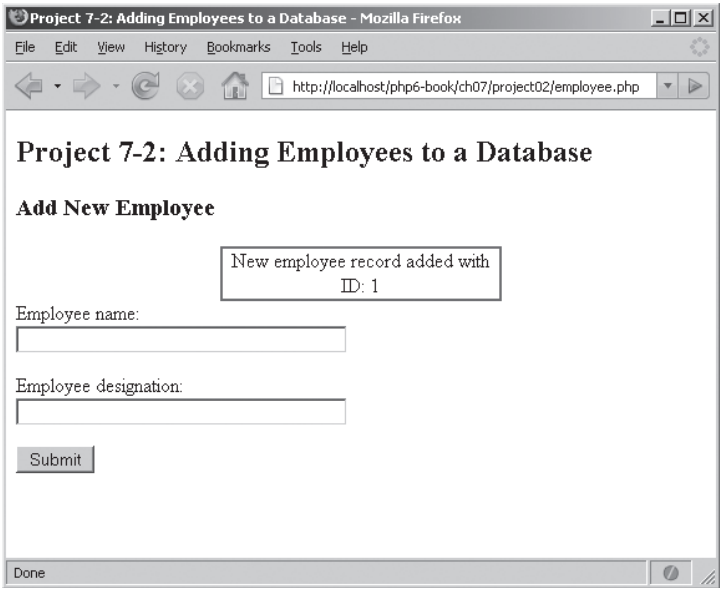

Figura 7-5 El resultado de una inserción correcta de un nuevo empleado en la base de datos

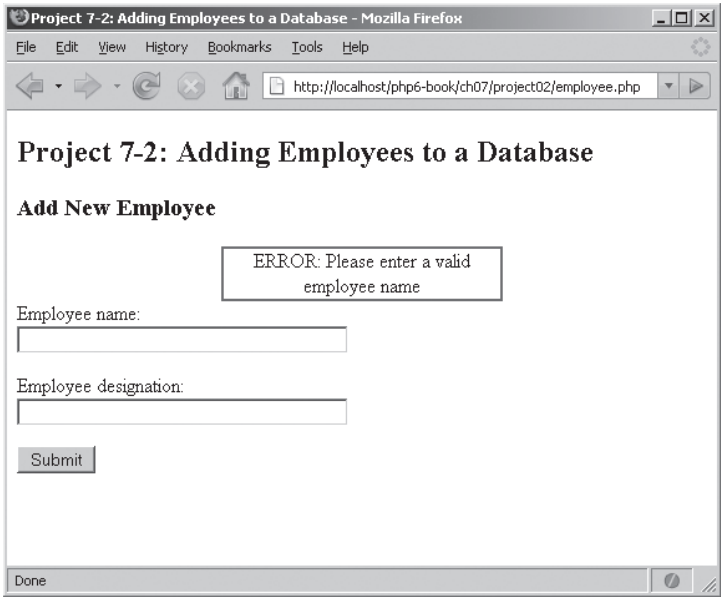

Figura 7-6 Los datos de salida después de enviar datos no válidos en el formulario Web

(*continúa*)

```
      text-align:center;
      margin-left:auto;
      margin-right:auto;
      width:40%;
      border: solid 2px green
    }
    table {
      border-collapse: collapse;
      width: 320px;
    }
    tr.heading {
      font-weight: bolder;
    }
    td {
      border: 1px solid black;
      padding: 0 0.5em;
    }
    </style>
  </head>
  <body>
    <h2>Proyecto 7-2: Añadir empleados a una base de datos</h2>
    <h3>Añade un Nuevo Empleado</h3>
<?php
    // intenta conectarse con la base de datos MySQL
    $mysqli = new mysqli("localhost", "usuario", "contra", "empleados");
    if ($mysqli === false) {
      die("ERROR: No fue posible conectarse con la base de datos. " .
mysqli_connect_error());
    }

    // si el formulario ha sido enviado
    // procesa los datos del formulario
    if (isset($_POST['submit'])) {
      // abre bloque de mensaje
      echo '<div id="message">';

      // recupera y verifica los valores de entrada
      $inputError = false;
      if (empty($_POST['emp_nombre'])) {
        echo 'ERROR: Por favor ingrese un nombre de empleado válido';
        $inputError = true;
      } else {
        $nombre = $mysqli->escape_string($_POST['emp_nombre']);
      }

      if ($inputError != true && empty($_POST['emp_puesto'])) {
        echo 'ERROR: Por favor ingrese un puesto válido';
        $inputError = true;
      } else {
```

```
        $puesto = $mysqli->escape_string($_POST['emp_puesto']);
      }

      // añade valores a la base de datos utilizando el consulta INSERT
      if ($inputError != true) {
      $sql = "INSERT INTO empleados (nombre, puesto)
              VALUES ('$nombre', '$puesto')";
        if ($mysqli->query($sql) === true) {
          echo 'Nuevo registro de empleado añadido con ID: ' . $mysqli
          ->insert_id;
        } else {
          echo "ERROR: No fue posible ejecutar el consulta: $sql. " .
$mysqli->error;
        }
      }

      // cierra bloque de mensaje
      echo '</div>';
    }
?>
    </div>

    <form action="empleados.php" method="POST">
      Nombre del empleado: <br />
      <input type="text" name="emp_nombre" size="40" />
      <p />
      Puesto del empleado: <br />
      <input type="text" name="emp_puesto" size="40" />
      <p />
      <input type="submit" name="submit" value="Enviar" />
    </form>

    <h3>Lista de empleados</h3>
<?php
    // obtiene registros
    // da formato como una tabla HTML
    $sql = "SELECT id, nombre, puesto FROM empleados";
    if($result = $mysqli->query($sql)) {
      if ($result->num_rows > 0) {
        echo "<table>\n";
        echo "  <tr class=\"heading\">\n";
        echo "    <td>ID</td>\n";
        echo "    <td>Nombre</td>\n";
        echo "    <td>Puesto</td>\n";
        echo "    </tr>\n";
        while ($row = $result->fetch_object()) {
          echo "  <tr>\n";
          echo "    <td>" . $row->id . "</td>\n";
```

(*continúa*)

```
            echo "    <td>" . $row->nombre . "</td>\n";
            echo "    <td>" . $row->puesto . "</td>\n";
            echo "  </tr>\n";
          }
          echo "</table>";
          $result->close();
      } else {
          echo "No hay empleados en la base de datos.";
        }
      } else {
        echo "ERROR: No fue posible ejecutar el consulta: $sql. " .
$mysqli->error;
      }

      // cierra conexión
      $mysqli->close();
?>
  </body>
</html>
```

Esta versión del script ahora incluye la consulta `SELECT`, que recupera los registros de la tabla de empleados como un objeto y los procesa utilizando un bucle. Estos registros son formados como una tabla HTML y presentados en la parte inferior de la página Web. Un efecto secundario útil de estas modificaciones es que los nuevos registros de empleados guardados mediante el formulario se reflejarán de inmediato en la tabla HTML una vez que se envíe el formulario.

La figura 7-7 muestra los datos de salida de esta versión modificada del script.

Utilizar la extensión SQLite de PHP

En este punto ya sabes conectar un script PHP a una base de datos MySQL para recuperar, añadir, editar y borrar registros. Sin embargo, MySQL no es el único juego en la ciudad; PHP 5.x también incluye soporte integrado para SQLite, opción eficiente y más ligera que MySQL. Esta sección aborda la base de datos SQLite y la extensión SQLite de PHP con gran detalle.

Introducción a SQLite

SQLite es una base de datos rápida y eficiente que ofrece una opción viable a MySQL, sobre todo para aplicaciones pequeñas y de tamaño mediano. Sin embargo, a diferencia de MySQL, que contiene una gran cantidad de componentes interrelacionados, SQLite está integrada en una sola biblioteca. También es significativamente más pequeña que MySQL, su línea de comandos pesa menos de 200 KB, y soporta todos los comandos SQL estándar con los que estás familiarizado. MySQL y SQLite también difieren en sus políticas de licencia: a diferencia de

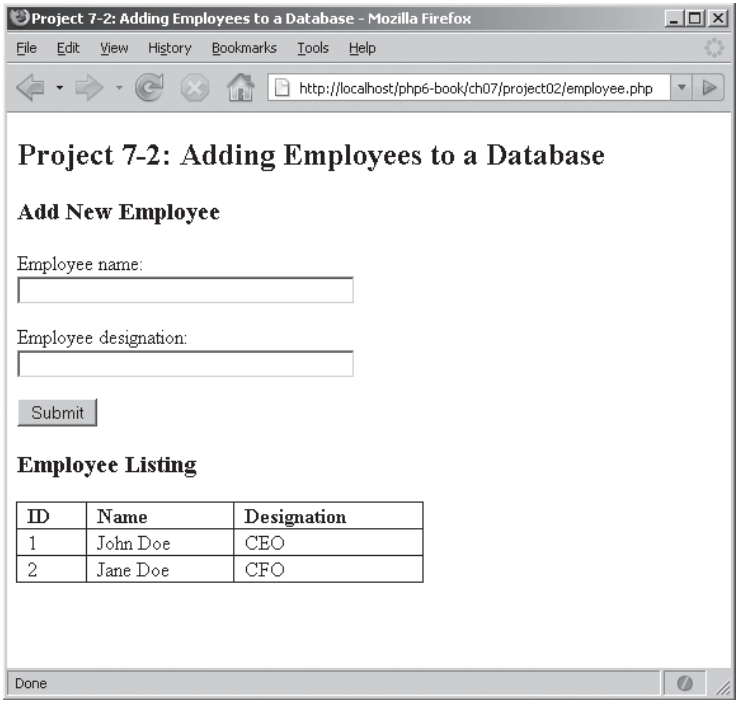

Figura 7-7 Página Web que muestra la lista de los empleados

MySQL, el código fuente de SQLite es completamente de dominio público, lo cual significa que los desarrolladores pueden utilizarlo y distribuirlo como les plazca, para productos tanto comerciales como no comerciales.

Sin embargo, el tamaño de SQLite disfraza su poder. La base de datos tiene una capacidad de soportar dos terabytes y puede tener un mejor rendimiento que MySQL en ciertas situaciones. En parte, esto se debe a razones estructurales: SQLite lee y escribe registros directamente del disco y por lo mismo ocupa menos recursos que MySQL, que opera sobre una arquitectura cliente-servidor que puede ser afectada por variables relacionadas con el nivel de la red.

NOTA

En el siguiente ejercicio, las palabras en negritas indican comandos que debes escribir en la línea de comando de SQLite. Los comandos pueden escribirse en mayúsculas o minúsculas. Antes de comenzar con el ejercicio asegúrate de haber instalado, configurado y probado la base de datos SQLite de acuerdo con las instrucciones que aparecen en el apéndice A de este libro.

SQLite da soporte a todas las declaraciones estándar de SQL que ya conoces y has llegado a amar en las dos secciones anteriores: `SELECT`, `INSERT`, `DELETE`, `UPDATE` y

`CREATE TABLE`. El siguiente ejemplo muestra el uso de SQLite replicando las tablas de la base de datos MySQL utilizadas en la sección anterior. Comienza iniciando el programa cliente de línea de comandos de SQLite y crea una nueva base de datos en el directorio en uso que lleve el nombre *musica.db*:

```
shell> sqlite musica.db
```

Si todo salió bien, verás un mensaje de bienvenida y un indicador SQL interactivo como el siguiente:

```
sqlite>
```

Ahora puedes comenzar a dictar declaraciones SQL desde este indicador. Comienza por crear una tabla que contenga la información de los artistas:

```
sqlite> CREATE TABLE artistas (
   ...> artista_id INTEGER NOT NULL PRIMARY KEY,
   ...> artista_nombre TEXT NOT NULL,
   ...> artista_pais TEXT NOT NULL
   ...>);
```

Contrariamente a MySQL, que ofrece un amplio rango de diferentes tipos de datos para sus campos, SQLite soporta sólo cuatro tipos, los cuales se muestran en la tabla 7-3.

Tipo de campo	Descripción
`INTEGER`	Campo numérico para firmar valores enteros
`REAL`	Campo numérico para valores de punto flotante
`TEXT`	Tipo cadena de caracteres
`BLOB`	Tipo binario

Tabla 7-3 Tipos de datos de SQLite

Lo que es más importante, SQLite es una base de datos "sin tipos". Esto significa que los campos de esta base de datos NO necesitan estar asociados a un tipo específico, e incluso cuando lo están, pueden almacenar valores de un tipo diferente al especificado. La única excepción a esta regla son los campos tipo `INTEGER PRIMARY KEY`: que son campos "autonuméricos" que generan identificadores numéricos únicos para cada registro de la tabla, de manera similar a los campos `AUTO_INCREMENT` de MySQL.

Con estos hechos en mente, continúa con el ejercicio y crea las restantes dos tablas utilizando estas declaraciones SQL:

```
sqlite> CREATE TABLE ratings (
   ...> rating_id INTEGER NOT NULL PRIMARY KEY,
```

```
   ...> rating_nombre TEXT NOT NULL
   ...> );

sqlite> CREATE TABLE canciones (
   ...> cancion_id INTEGER NOT NULL PRIMARY KEY,
   ...> cancion_titulo TEXT NOT NULL,
   ...> ex_cancion_artista INTEGER NOT NULL,
   ...> ex_cancion_rating INTEGER NOT NULL
   ...> );
```

Ahora alimenta las tablas con registros:

```
sqlite> INSERT INTO artistas (artista_id, artista_nombre, artista_pais)
   ...> VALUES ('1', 'Aerosmith', 'US');
sqlite> INSERT INTO artistas (artista_nombre, artista_pais)
   ...> VALUES ('Abba', 'SE');

sqlite> INSERT INTO ratings (rating_id, rating_nombre)
   ...> VALUES (4, 'Bueno');
sqlite> INSERT INTO ratings (rating_id, rating_nombre)
   ...> VALUES (5, 'Excelente');

sqlite> INSERT INTO canciones (cancion_titulo, ex_cancion_artista,
        ex_cancion_rating)
   ...> VALUES ('Janie''s Got a Gun', 1, 4);
sqlite> INSERT INTO canciones (cancion_titulo, ex_cancion_artista,
        ex_cancion_rating)
   ...> VALUES ('Crazy', 1, 5);
```

NOTA

El archivo de código para este libro tiene una lista completa con las declaraciones SQL `INSERT` necesarias para alimentar las tres tablas utilizadas en este ejercicio. Ejecuta esas declaraciones y termina de construir las tablas antes de proseguir.

Una vez que hayas terminado, prueba la declaración `SELECT`:

```
sqlite> SELECT artista_nombre, artista_pais FROM artistas
   ...> WHERE artista_pais = 'US'
   ...> OR artista_pais = 'UK';
Aerosmith|US
Timbaland|US
Take That|UK
Girls Aloud|UK
```

SQLite soporta por completo las fusiones SQL; he aquí un ejemplo:

```
sqlite> SELECT cancion_titulo, artista_nombre, rating_nombre
   ...> FROM canciones, artistas, ratings
   ...> WHERE canciones.ex_cancion_artista = artista.artista_id
   ...> AND canciones.ex_canciones_rating = ratings.rating_id
```

```
   ...> AND ratings.rating_id >= 4
   ...> AND artistas.artista_pais != 'US';
En Las Delicious|Cubanismo|Fantastic
Pray|Take That|Good
SOS|Abba|Fantastic
Dancing Queen|Abba|Good
```

Recuperar datos

Recuperar datos de una base SQLite con PHP no es muy diferente de recuperarlos de una base MySQL. El siguiente script PHP muestra el proceso utilizando la base de datos *musica.db* creada en la sección anterior:

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");

// intenta ejecutar el consulta
// reitera sobre una colección de resultados
// presenta cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $sqlite->query($sql)) {
  if($result->numRows() > 0) {
    while($row = $result->fetch()) {
      echo $row[0] . ":" . $row[1] . "\n";
    }
  } else {
    echo "No se encontraron registros que coincidieran con los criterios
del consulta.";
  }
} else{
  echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

El script realiza las siguientes acciones:

1. Abre el archivo de base de datos, para realizar las operaciones, al inicializar una instancia de la clase SQLiteDatabase y transmitiendo al constructor del objeto la ruta de acceso completa a la base de datos. Si este archivo de base de datos no es localizado, se creará uno vacío con el nombre proporcionado (siempre y cuando el script tenga privilegios de escritura sobre el directorio correspondiente).

2. El siguiente paso consiste en crear y ejecutar la consulta SQL. Esto se realiza invocando el método `query()` del objeto SQLiteDatabase y transmitiéndole el consulta que se va a ejecutar. Dependiendo de si el consulta fue exitoso o fracasó, la función regresa un valor verdadero o falso; en caso de fallo, el código de error correspondiente a la razón de la falla puede obtenerse invocando el método `lastError()` del objeto SQLiteDatabase. La función `sqlite_error_string()` convierte este código de error en un mensaje comprensible para los humanos.

TIP

Hay una semejanza cercana entre estos pasos y los que seguiste para utilizar la extensión MySQL en la sección anterior.

3. Por otra parte, si la consulta se ejecutó correctamente y regresa uno o más registros, el valor de retorno del método `query()` es otro objeto, este último será una instancia de la clase SQLiteResult. Este objeto representa la colección de resultados regresada por el consulta, y expone la consulta para procesar registros individuales en la colección de resultados.

   Este script utiliza el método `fetch()`, que regresa el siguiente resultado de la colección de resultados como una matriz cada vez que se invoca. Utilizado en un bucle, este método proporciona una manera conveniente de hacer reiteraciones sobre la colección de resultados, un registro a la vez. Los campos individuales del registro pueden ser accesados como elementos de la matriz, utilizando ya sea el índice o el nombre del campo. La cantidad de resultados en la colección de éstos puede recuperarse a través del método `numRows()` del objeto SQLiteResult.

4. Una vez que la colección de resultados entera ha sido procesada y no restan operaciones por ejecutarse en el archivo de base de datos, es buena idea cerrar el manejador de la base de datos para liberar la memoria que ocupa, lo cual se realiza destruyendo el objeto SQLiteDatabase.

Recuperar registros como matrices y objetos

El método `fetch()` del objeto SQLiteResult acepta un modificador adicional, que controla la manera en que se recuperan los elementos de la colección de resultados. Este modificador puede ser cualquiera de los valores `SQLITE_NUM` (para recuperar cada registro como una matriz numérica indexada), `SQLITE_ASSOC` (para regresar cada registro como una matriz asociativa) o `SQLITE_BOTH` (para regresar cada registro de ambas maneras, como una matriz numérica indexada y como una matriz asociativa, además de la opción por defecto). He aquí un ejemplo, que muestra estos modificadores en acción y produce una serie de datos de salida equivalente al ejemplo anterior:

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");
// intenta ejecutar el consulta
// presenta registros utilizando diferentes estilos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if($result = $sqlite->query($sql)) {

  // recupera registros como una matriz numérica
  $row = $result->fetch(SQLITE_NUM);
  echo $row[0] . ":" . $row[1] . "\n";

  // recupera registros como una matriz asociativa
  $row = $result->fetch(SQLITE_ASSOC);
  echo $row['artista_id'] . ":" . $row['artista_nombre'] . "\n";

  // recupera registros como un objeto
  $row = $result->fetchObject();
  echo $row->artista_id . ":" . $row->artista_nombre . "\n";

} else {
  echo "ERROR: no fue posible ejecutar $sql. " . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

También es posible regresar cada registro como un objeto, reemplazando `fetch()` con el método `fetchObject()`. He aquí un ejemplo equivalente al anterior, sólo que en lugar de recuperar valores de campo como elementos de una matriz, lo hace como propiedades de un objeto:

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");
// intenta ejecutar el consulta
// reitera sobre una colección de resultados
// presenta cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if($result = $sqlite->query($sql)) {
  if ($result->numRows() > 0) {
    while ($row = $result->fetchObject()) {
      echo $row->artista_id . ":" . $row->artista_nombre . "\n";
```

```
    }
  } else {
    echo "No se encontraron registros que coincidieran con los criterios
del consulta.";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

Una característica importante de la extensión SQLite es su capacidad de recuperar *todos* los registros de una colección de resultados al mismo tiempo como un conjunto de matrices anidadas, a través del método `fectchAll()` del objeto SQLiteResult. Un bucle `foreach` puede reiterar sobre esta colección anidada, recuperando un registro tras otro. El siguiente ejemplo muestra este procedimiento en acción:

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");

// intenta ejecutar el consulta
// reitera sobre una colección de resultados
// presenta cada registro y sus campos
// datos de salida: "1:Aerosmith \n 2:Abba \n ..."
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $sqlite->query($sql)) {
  $data = $result->fetchAll();
  if (count($data) > 0) {
    foreach ($data as $row) {
      echo $row[0] . ":" . $row[1] . "\n";
    }
  } else {
    echo "No se encontraron registros que coincidieran con los criterios
del consulta.";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

PRECAUCIÓN

El método `fetchAll()` regresa la colección de resultados completa como un conjunto de matrices anidadas, y todos ellos se almacenan en la memoria de la computadora hasta que termina el proceso. Para no agotar la memoria, no utilices este método si es posible que tu consulta regrese un gran número de registros.

Añadir y modificar datos

Para consultas que no regresan una colección de resultados, como `INSERT`, `UPDATE` y `DELETE`, la extensión SQLite ofrece el método `queyExec()`. El siguiente ejemplo lo muestra en acción, añadiendo un nuevo registro a la tabla *artistas*.

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");

// intenta ejecutar el consulta
// añade un nuevo registro
// datos de salida: "Nuevo artista con el id:8 ha sido añadido."
$sql = "INSERT INTO artistas (artista_nombre, artista_pais) VALUES
('James Blunt', 'UK')";
if (sqlite->consultaExec($sql) == true) {
  echo 'Nuevo artista con el id:' . $sqlite->lastInsertRowid() . 'ha sido
añadido.';
} else {
  echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

Dependiendo de si el consulta se ejecutó con éxito o no, la función `consultaExec()` regresa un valor verdadero o falso; es fácil revisar este valor de respuesta y mostrar un mensaje de éxito o de falla. Si el registro fue insertado en una tabla con un campo `INTEGER PRIMARY KEY`, SQLite automáticamente asignará al registro un número de identificación único. Este número puede ser recuperado con el método `lastInsertRowid()` del objeto SQLiteDatabase, como se mostró en el ejemplo anterior.

PRECAUCIÓN

Para que funcionen los comandos `INSERT`, `UPDATE` y `DELETE`, recuerda que el script PHP *debe* tener privilegios de escritura para el archivo de base de datos SQLite.

Cuando realizas un consulta `UPDATE` o `DELETE`, la cantidad de filas afectadas por éste también puede ser recuperada a través del método `changes()` del objeto SQLiteDatabase. El siguiente ejemplo lo muestra, actualizando los ratings de las canciones en la base de datos y regresando la cuenta de los registros afectados:

```
<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('musica.db') or die ("No fue posible abrir
la base de datos");

// intenta ejecutar el consulta
// actualiza registro
// datos de salida: "3 fila(s) actualizadas."
$sql = "UPDATE canciones SET ex_cancion_rating = 4 WHERE ex_cancion_
rating = 3";
if (sqlite->consultaExec($sql) == true) {
  echo $sqlite->changes() . 'fila(s) actualizadas.';
} else {
  echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>
```

Manejo de errores

Ambos métodos, `query()` y `consultaExec()`, regresan un valor falso si ocurre un error durante la preparación o ejecución del consulta. Es fácil verificar el valor de retorno de estos métodos y recuperar el código correspondiente al error invocando el método `lastError()` del objeto SQLiteDatabase. Desafortunadamente, este código de error no es de gran utilidad por sí mismo; así que, para obtener una descripción literal de lo que sucedió, se debe envolver la invocación `lastError()` en la función `sqlite_error_string()`, como se hizo en la mayoría de los ejemplos anteriores.

Prueba esto 7-3 Crear una lista personal de pendientes

Ahora que ya tienes una idea del manejo de SQLite, ¿qué tal si lo utilizas en una aplicación práctica?: una lista personal de pendientes que puedas consultar y actualizar a través del explorador Web. Esta lista de pendientes te permitirá ingresar tareas y fechas de vencimiento, asignar prioridades a las tareas, editar sus descripciones y marcarlas cuando hayan sido com-

pletadas. Es un poco más complicada que las aplicaciones que has hecho hasta ahora, porque incluye varias partes dinámicas; sin embargo, si has puesto atención hasta este punto, no será muy difícil que comprendas lo que está sucediendo.

Para comenzar, crea una nueva base de datos SQLite llamada *pendientes.db* y añade una tabla vacía para contener la descripción de las tareas y las fechas:

```
shell> sqlite pendientes.db
sqlite> CREATE TABLE tareas (
   ...> id INTEGER PRIMARY KEY,
   ...> nombre TEXT NOT NULL,
   ...> vencimiento TEXT NOT NULL,
   ...> prioridad TEXT NOT NULL
   ...> );
```

Esta tabla tiene cuatro campos: para el ID del registro, el nombre de la tarea, la fecha de vencimiento de ésta y la prioridad que tiene.

El siguiente paso consiste en crear un formulario Web para añadir nuevas tareas a la base de datos (*guarda.php*):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-3: Crear una lista personal de pendientes</title>
    <style type="text/css">
    div#message{
      text-align:center;
      margin-left:auto;
      margin-right:auto;
      width:40%;
      border: solid 2px green
    }
    </style>
  </head>
  <body>
    <h2>Proyecto 7-3: Crear una lista personal de pendientes</h2>
    <h3>Añade una Nueva Tarea</h3>

    <?php
    // si el formulario no ha sido enviado
    // genera un nuevo formulario
    if (!isset($_POST['submit'])) {
    ?>

    <form method="post" action="guarda.php">
      Descripción: <br />
      <input type="text" name="nombre" />
```

```
      <p>
      Fecha de vencimiento (dd/mm/aaaa): <br />
      <input type="text" name="dd" size="2" />
      <input type="text" name="mm" size="2" />
      <input type="text" name="aa" size="4" />
      <p>
      Prioridad: <br />
      <select name="prioridad">
        <option name="Alta">Alta</option>
        <option name="Media">Media</option>
        <option name="Baja">Baja</option>
      </select>
      <p>
      <input type="submit" name="sumbit" value="Guardar" />
    </form>

    <?php
    } else {
      // si el formulario ya fue enviado
      // intenta establecer conexión con la base de datos
      $sqlite = new SQLiteDataBase('pendientes.db') or die ("No fue
posible abrir la base de datos");

      // verifica y limpia los datos de entrada
      $nombre = !empty($_POST['nombre']) ? sqlite_escape_string
($_POST['nombre']) : die('ERROR: Se requiere el nombre de la tarea');
      $dd = !empty($_POST['dd']) ? sqlite_escape_string (int)
($_POST['dd']) : die('ERROR: Se requiere fecha de vencimiento');
      $mm = !empty($_POST['mm']) ? sqlite_escape_string (int)
($_POST['mm']) : die('ERROR: Se requiere fecha de vencimiento');
      $aa = !empty($_POST['aa']) ? sqlite_escape_string (int)
($_POST['aa']) : die('ERROR: Se requiere fecha de vencimiento');
      $fecha = checkdate($mm, $dd, $aa) ? mktime(0, 0, 0, $mm, $dd, $aa)
: die('ERROR: La fecha de vencimiento es inválida');
      $prioridad = !empty($_POST['prioridad']) ? sqlite_escape_string
($_POST['prioridad']) : die('ERROR: Se requiere la prioridad de la
tarea');

      // intenta ejecutar el consulta
      // añade nuevo registro
      $sql = "INSERT INTO tareas (nombre, vencimiento, prioridad) VALUES
('$nombre', '$vencimiento', '$prioridad')";
      if ($sqlite->consultaExec($sql) == true){
        echo '<div id="message"> El registro ha sido añadido con éxito a
la base de datos.</div>';
      } else {
        echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
```

(*continúa*)

```
        }

        // cierra conexión
        unset($sqlite);
      }
      ?>

  </body>
</html>
```

La figura 7-8 muestra el formulario Web.

Cuando el usuario envía este formulario, primero se validan los datos para asegurar que estén presentes todos los campos requeridos. Los datos ingresados en los tres campos correspondientes a la fecha también son verificados con la función PHP `checkdate()` para asegurar que los tres juntos formen una fecha válida. Después, se limpian los datos de entrada al pasarlos por la función `sqlite_escape_string()`, que elimina los caracteres especiales de los datos de entrada automáticamente y los guarda en la base de datos utilizando el consulta `INSERT`.

La figura 7-9 muestra el resultado de añadir con éxito una nueva tarea a la base de datos.

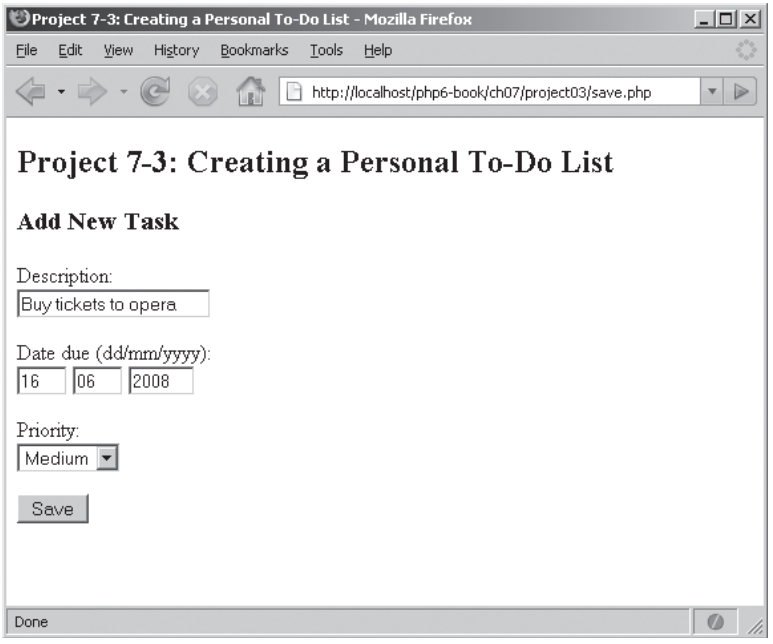

Figura 7-8 Formulario Web para añadir tareas a la base de datos

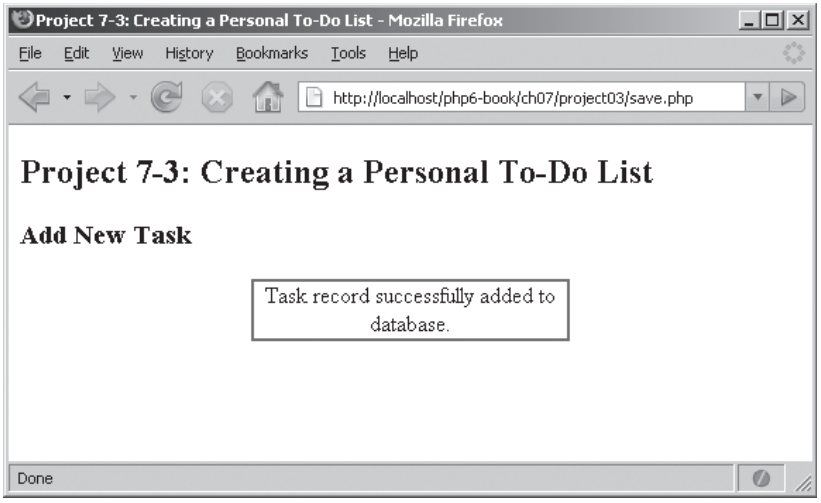

Figura 7-9 El resultado de añadir un nuevo pendiente a la base de datos

Eso es suficiente para ingresar tareas a la lista de pendientes. ¿Qué tal si ahora los vemos? Como probablemente lo habrás adivinado, es cuestión de utilizar una consulta `SELECT` para recuperar todos los registros de la base de datos y luego dar formato a la información resultante de manera que sea apropiada para desplegarse en una página Web. He aquí el código (*lista.php*):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-3: Crear una lista de pendientes personales</title>
    <style type="text/css">
    div#message{
      text-align:center;
      margin-left:auto;
      margin-right:auto;
      width:60%;
      border: solid 2px green
    }
    table{
    border-collapse: collapse;
    width: 500px;
    }
    tr.heading{
    font-weight: bolder;
     }
```

(*continúa*)

```
      td{
     border: 1px solid black;
     padding: 1em;
      }
      tr.high {
        background: #cc1111;
      }
      tr.medium {
        background: #00aaaa;
      }
      tr.low {
        background: #66dd33;
      }
      a {
         color: black;
         border: outset 2px black;
         text-decoration: none;
         padding: 3px;
      }
      </style>
  </head>
  <body>
    <h2>Proyecto 7-3: Crear una lista de pendientes personales</h2>
    <h3>Lista de Tareas</h3>

<?php
// intenta establecer conexión con la base de datos
$sqlite = new SQLiteDataBase('pendientes.db') or die ("No fue posible
abrir la base de datos");

// obtiene registros
// como una tabla HTML
$sql = "SELECT id, nombre, vencimiento, prioridad FROM tareas ORDER BY
vencimiento";
if ($result = $sqlite->query($sql)) {
  if ($result->numRows() > 0) {
    echo "<table>\n";
    echo "  <tr class=\"heading\">\n";
    echo "    <td>Descripción</td>\n";
    echo "    <td>Fecha de vencimiento</td>\n";
    echo "    <td></td>\n";
    echo "  </tr>\n";
    while($row = $result->fetchObject()) {
      echo "  <tr class=\"" . strtolower($row->priority) . "\">\n";
      echo "    <td>" . $row->nombre . "</td>\n";
      echo "    <td>" . date('d M Y', $row->vencimiento) . "</td>\n";
      echo "    <td><a href=\"marca.php?id=" . $row->id . "\"> Marcar
como finalizada</a></td>\n";
      echo "  </tr>\n";
```

```
    }
    echo "</table>";
  } else {
    echo '<div id="message"> No hay tareas en la base de datos.</div>';
  }
} else {
  echo 'ERROR: No fue posible ejecutar consulta: $sql.' . sqlite_error_
string($sqlite->lastError());
}

// cierra conexión
unset($sqlite);
?>

  <p/>
  <a href="guarda.php">Añadir una nueva tarea</a>

  </body>
</html>
```

No hay nada extraño aquí: el script ejecuta la consulta `SELECT` con el método `exec()`, y luego itera sobre la colección de resultados, presentando cada registro encontrado como una fila de una tabla HTML. Advierte que dependiendo del campo *prioridad* de cada registro, la fila de la tabla HTML correspondiente tiene un color diferente: rojo, verde o azul.

La figura 7-10 muestra un ejemplo de los datos de salida.

Verás algo más en la figura: cada elemento de la lista de pendientes viene con una opción `'Marcar como Finalizada'`. Esta opción apunta hacia otro script PHP, *marca.php*, que es responsable de eliminar el registro correspondiente de la base de datos. Mira con atención el URL que apunta a *marca.php*, y verás que el ID del registro también es transmitido, como la variable `$id`. Dentro de *marca.php*, este ID será accesado a través de la matriz `$_GET`, como `$_GET['id']`.

¿Qué hace *marca.php*? Nada muy complejo, simplemente utiliza el ID del registro transmitido por `$_GET` para formular y ejecutar un consulta `DELETE`, que elimina el registro de la base de datos. He aquí el código (*marca.php*):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-3: Crear una lista de pendientes personales</title>
    <style type="text/css">
    div#message {
      text-align:center;
      margin-left:auto;
      margin-right:auto;
      width:40%;
      border: solid 2px green
```

(*continúa*)

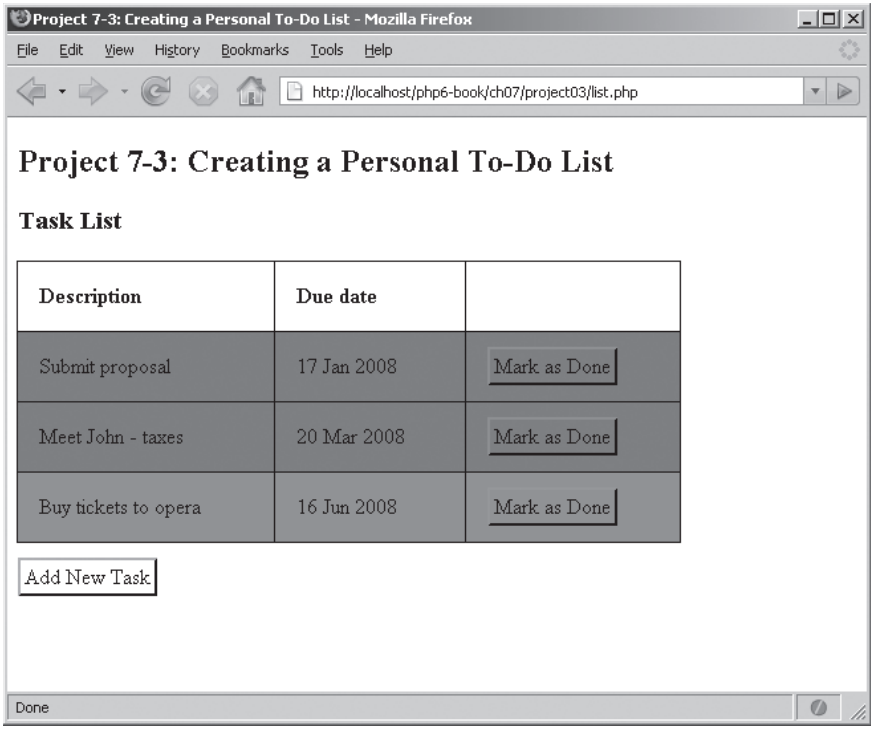

Figura 7-10 Página Web que muestra los elementos de la lista de pendientes, ordenados por fecha de vencimiento

```
    }
    </style>
  </head>
  <body>
    <h2>Proyecto 7-3: Crear una lista de pendientes personales</h2>
    <h3>Elimina la Tarea Terminada</h3>

    <?php
    if (isset($_GET['id'])) {
      // intenta establecer conexión con la base de datos
      $sqlite = new SQLiteDataBase('pendientes.db') or die ("No fue
posible abrir la base de datos");

      // verifica y limpia los datos de entrada
      $id = !empty($_GET['id']) ? sqlite_escape_string((int)$_GET['id'])
: die('ERROR: Se requiere el ID de la tarea');

      // elimina registro
      $sql = "DELETE FROM tareas WHERE id = '$id'";
```

```
      if ($sqlite->consultaExec($sql) == true) {
        echo '<div id="message">Registro de la tarea eliminado
exitosamente de la base de datos.</div>';
      } else {
        echo "ERROR: No fue posible ejecutar $sql." . sqlite_error_
string($sqlite->lastError());
      }

      // cierra conexión
      unset($sqlite);
    } else {
      die ('ERROR: Se requiere el ID de la tarea');
    }
    ?>

  </body>
</html>
```

La figura 7-11 muestra los datos de salida al eliminar de manera correcta una tarea de la base de datos.

Y ahora, cuando revisas *lista.php*, tu lista de pendientes ya no mostrará el elemento que has marcado como terminado. Sencillo, ¿no es cierto?

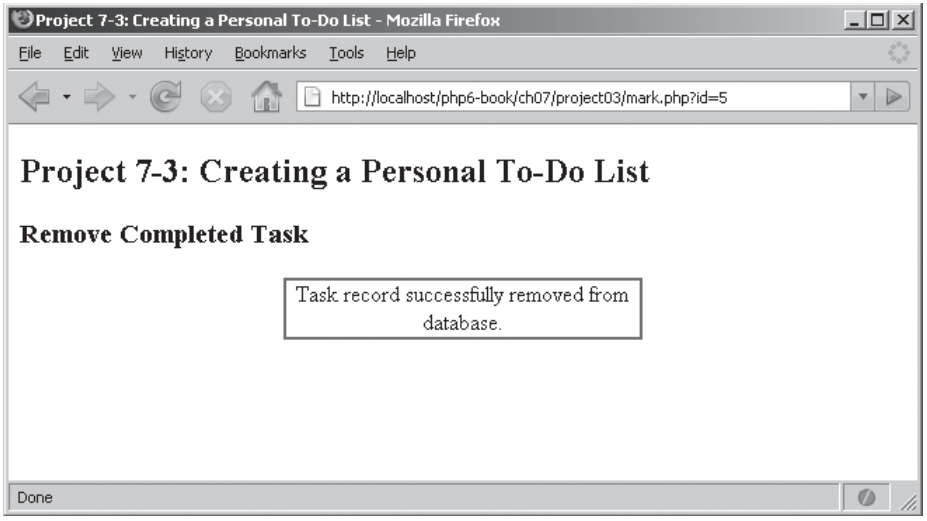

Figura 7-11 El resultado de marcar una tarea como realizada

Utilizar las extensiones PDO de PHP

En las secciones anteriores aprendiste a integrar tus aplicaciones PHP con las bases de datos MySQL y SQLite. Como ya habrás notado, las extensiones MySQLi y SQLite utilizan diferentes nombres de métodos para realizar su trabajo; como resultado, conmutar de una base de datos a otra implica, en esencia, volver a escribir todo el código de base de datos para utilizar los nuevos métodos. Por ello PHP ofrece una extensión neutral para bases de datos: objetos de datos de PHP (PDO), que brinda gran portabilidad y puede reducir de manera significativa el esfuerzo que implica conmutar de un sistema de base de datos a otro.

La siguiente sección aborda la extensión PDO con gran detalle, proporcionando información sobre la manera en que se utiliza para conectarse a diferentes sistemas de bases de datos, realizar consultas, procesar colecciones de resultados, además de manejar errores de consultas y conexión. La mayor parte de los ejemplos dan por hecho que se trabaja con una base de datos MySQL; sin embargo, como verás, los programas basados en PDO requieren mínimas modificaciones para trabajar con otros sistemas de bases de datos, incluyendo SQLite.

Recuperar datos

PDO trabaja proporcionando un conjunto estándar de funciones para realizar operaciones comunes de base de datos, como conexión, consultas, procesamiento de colecciones de resultados y manejo de errores; internamente traduce estas funciones a las invocaciones API nativas comprensibles para la base de datos en uso. Para mostrar la manera en que funciona, examina el siguiente ejemplo, que ejecuta una consulta `SELECT` y despliega los registros encontrados:

```
<?php
// intenta establecer conexión
try {
   $pdo = new PDO('mysql:dbname=musica;host=localhost', 'usuario',
'contra');
} catch (PDOException $e) {
   die("Error: No fue posible conectar: " . $e->getMessage());
}

// crea y ejecuta consulta SELECT
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $pdo->query($sql)) {
  while($row = $result->fetch()) {
    echo $row[0] . ":" . $row[1] . "\n";
  }
} else {
  echo "ERROR: No fue posible ejecutar $sql. " . print_r($pdo
->errorInfo());
```

```
}

// cierra conexión
unset($pdo);
?>
```

Como lo muestra este ejemplo, utilizar PDO para obtener datos de una base implica pasos semejantes a los que has visto en secciones previas:

1. El primer paso consiste en inicializar una instancia de la clase PDO y pasar al constructor del objeto tres argumentos: una cadena con el nombre del origen de datos (DSN, Data Source Name), indicando el tipo de base de datos al que se va a conectar, además de otras opciones específicas de la base, un nombre de usuario válido reconocido por la base en cuestión y su correspondiente contraseña. La cadena DSN varía de una a otra base de datos; por lo general puedes obtener el formato exacto para esta cadena de la documentación de la base que estés utilizando.

   La tabla 7-4 muestra algunos formatos de cadenas DSN comunes.

   Si el intento de conexión fracasa, se generará una excepción; esta excepción puede ser recogida y manejada utilizando el mecanismo de manejo de excepciones de PHP (puede obtenerse más información sobre el manejo de excepciones en el capítulo 10 de este libro).

2. Suponiendo que la conexión fue correcta, el siguiente paso consiste en formular una consulta SQL y ejecutarla utilizando el método `query()` de PDO. El valor de regreso de este método es una colección de resultados, representada por el objeto PDOStatement. El contenido de la colección de resultados puede procesarse utilizando el método `fetch()`

Base de datos	Cadena DSN
MySQL	`'mysql:host=`*`host`*`; port=`*`puerto`*`;dbname=`*`db`*`'`
SQLite	`'sqlite:`*`ruta/a/archivo/basededatos`*`'`
PostgreSQL	`'pgsql:host=`*`host`*` port=`*`puerto`*` dbname=`*`db`*` usuario=`*`usuario`*` password=`*`contra`*`'`
Oracle	`'oci:dbname=//`*`host`*`:`*`puerto`*`/`*`db`*`'`
Firebird	`'firebird:Usuario=`*`usuario`*`;Password=`*`contra`*`; Database=`*`db`*`; DataSource=`*`host`*`;Port=`*`puerto`*`'`

Tabla 7-4 Cadenas DSN comunes

del objeto, que regresa el siguiente registro en la colección de resultados como una matriz (tanto asociativa como indexada). Es posible acceder a los campos individuales del registro como elementos de la matriz en un bucle, utilizando el índice del campo o su nombre.

Pregunta al experto

P: ¿Cómo calculo el número de registros en mi colección de resultados con PDO?

R: A diferencia de las extensiones MySQL y SQLite, PDO no ofrece métodos ni propiedades integrados para recuperar directamente el número de registros en una colección de resultados. Esto se debe a que no todos los sistemas de bases de datos regresan esta información, por lo que PDO no puede proporcionar esta información de manera portátil. Sin embargo, si necesitas esta información, el manual PHP recomienda ejecutar la declaración `SELECT COUNT(*)...` para obtenerla, con la consulta deseada y recuperar el primer campo de la colección de resultados, que contendrá el número de registros que coinciden con la consulta. Para mayor información, revisa en el análisis de www.php.net/manual/en/function.PDOStatement-rowCount.php.

El método `fetch()` del objeto PDOStatement acepta un modificador adicional, que controla la manera en que se realiza la búsqueda en la colección de resultados. Algunos valores aceptados por este modificador se muestran en la tabla 7-5.

Modificador	Lo que hace
`PDO::FETCH_NUM`	Regresa cada registro como una matriz numérica indexada
`PDO::FETCH_ASSOC`	Regresa cada registro como una matriz asociativa cuya clave es el nombre de campo
`PDO::FETCH_BOTH`	Regresa cada registro de ambas maneras, como una matriz numérica indexada y como una matriz asociativa (el valor por defecto)
`PDO::FETCH_OBJ`	Regresa cada registro como un objeto con propiedades correspondientes a los nombres de campo
`PDO::FETCH_LAZY`	Regresa cada registro como una matriz numérica indexada, como una matriz asociativa y como un objeto

Tabla 7-5 Modificadores del método `fetch()` de PDO

El siguiente ejemplo muestra estos modificadores en acción:

```
<?php
// intenta establecer una conexión
try {
   $pdo = new PDO('mysql:dbname=musica;host=localhost', 'usuario',
'contra');
} catch (PDOException $e) {
   die("Error: No fue posible conectar: " . $e->getMessage());
}

// crea y ejecuta consulta SELECT
$sql = "SELECT artista_id, artista_nombre FROM artistas";
if ($result = $pdo->query($sql)){

  // regresa registro como matriz numérica
  $row = $result->fetch(PDO::FETCH_NUM);
  echo $row[0] . ":" . $row[1] . "\n";

  // regresa registro como matriz asociativa
  $row = $result->fetch(PDO::FETCH_ASSOC);
  echo $row['artista_id'] . ":" . $row['artista_nombre'] . "\n";

  // regresa registro como un objeto
  $row = $result->fetch(PDO::FETCH_OBJ);
  echo $row->artista_id . ":" . $row->artista_nombre . "\n";

  // regresa registro utilizando una combinación de estilos
  $row = $result->fetch(PDO::FETCH_LAZY);
  echo $row['artista_id'] . ":" . $row->artista_nombre . "\n";

} else {
  echo "ERROR: No fue posible ejecutar $sql. " . print_r($pdo-
>errorInfo());
}

// cierra conexión
unset($pdo);
?>
```

Añadir y modificar datos

PDO también facilita la ejecución de consultas `INSERT`, `UPDATE` y `DELETE` con su método `exec()`. Este método, que está diseñado para instrucciones que de alguna manera modifican la base de datos, regresa la cantidad de registros afectados por el consulta. He aquí un ejemplo de su uso para insertar y eliminar un registro:

```
<?php
// intenta establecer una conexión
try {
   $pdo = new PDO('mysql:dbname=musica;host=localhost', 'usuario',
'contra');
} catch (PDOException $e) {
   die("ERROR: No fue posible conectar: " . $e->getMessage());
}

// crea y ejecuta consulta INSERT
$sql = "INSERT INTO artistas (artista_nombre, artista_pais) VALUES
('Luciano Pavarotti', 'IT')";
$ret = $pdo->exec($sql);
if ($ret === false) {
  echo "ERROR: No fue posible ejecutar $sql. " . print_r($pdo
->errorInfo());
} else {
  $id = $pdo->lastInsertId();
  echo 'Nuevo artista con id: ' . $id . 'ha sido añadido.';
}

// crea y ejecuta un consulta DELETE
$sql = "DELETE FROM artistas WHERE artista_pais = 'IT'";
$ret = pdo->exec($sql);
if ($ret === false) {
  echo "ERROR: No fue posible ejecutar $sql. " . print_r($pdo
->errorInfo());
} else {
  echo 'Eliminados ' . $ret . ' registros.';
}

// cierra conexión
unset($pdo);
?>
```

Este código utiliza el método `exec()` dos veces, primero para insertar un nuevo registro y luego para eliminar registros que coincidan con una condición específica. Si la consulta transmitida a `exec()` falla, `excec()` regresa un valor falso; de lo contrario, regresa la cantidad de registros afectados por la consulta. Advierte también que el script utiliza el método `lastInsertId()` del objeto PDO, el cual regresa el ID generado por el último comando `INSERT` en caso de que la tabla contenga un campo de autoincremento.

TIP

Si no hay registros afectados por la ejecución de la consulta realizada por el método `exec()`, éste regresará un valor igual a cero. No confundas este valor con el valor booleano "false" (falso) regresado por `exec()` cuando falla la consulta. Para evitar confusiones, el manual de PHP recomienda siempre probar el valor de retorno de `exec()` con el operador === en lugar de utilizar el operador ==.

Utilizar declaraciones preparadas

PDO también da soporte a declaraciones preparadas, mediante sus métodos `prepare()` y `execute()`. El siguiente ejemplo lo ilustra, utilizando una declaración preparada para añadir varias canciones a la base de datos:

```
<?php
// define los valores que serán insertados
$canciones = array(
  array('Voulez-Vous', 2, 5),
  array('Take a Chance on Me', 2, 3),
  array('I Have a Dream', 2, 4),
  array('Thank You For The Music', 2, 4),
);

// intenta establecer una conexión
try {
   $pdo = new PDO('mysql:dbname=musica;host=localhost', 'usuario',
'contra');
} catch (PDOException $e) {
   die("ERROR: No fue posible conectar: " . $e->getMessage());
}

// crea y ejecuta consulta SELECT
sql = "INSERT INTO canciones (cancion_titulo, ex_cancion_artista, ex_
cancion_rating) VALUES (?, ?, ?)";
if ($stmt = $pdo->prepare($sql)) {
  foreach ($canciones as $s) {
    $stmt->bindParam(1, $s[0]);
    $stmt->bindParam(2, $s[1]);
    $stmt->bindParam(3, $s[2]);
    if ($stmt->execute()) {
      echo "Nueva canción con id: " . $pdo->lastInsertId() . "ha sido
añadida. \n";
    } else {
      echo "ERROR: No fue posible ejecutar consulta: $sql. " . print_r
($pdo->errorInfo());
    }
  }
} else {
  echo "ERROR: No fue posible preparar consulta: $sql. " . print_r($pdo
->errorInfo());
}

// cierra conexión
unset($pdo);
?>
```

Si comparas este último script con uno similar de MySQLi en las secciones anteriores, verás una similitud muy marcada. Como antes, este script también define una consulta SQL modelo que contiene una declaración `INSERT` formada por contenedores en lugar de valores, y luego convierte este modelo en una declaración preparada utilizando el método `prepare()` del objeto PDO.

En caso de tener éxito, `prepare()` regresa un objeto PDOStatement que representa la declaración preparada. Los valores que serán interpolados en la declaración se unen entonces a los contenedores invocando repetidamente el método `bindParam()` del objeto PDOStatement con dos argumentos, la posición del contenedor y la variable a la que será unida. Una vez que las variables están unidas, el método `execute()` del objeto PDOStatement se ocupa de ejecutar la declaración preparada con los valores correctos.

NOTA

Cuando utilices declaraciones preparadas con PDO, es importante considerar que los beneficios de este tipo de declaraciones sólo estarán disponibles si la base de datos con la que trabajas soporta estas declaraciones de forma nativa. Para bases de datos que no soportan las declaraciones preparadas, PDO convertirá internamente las invocaciones para `prepare()` y `execute()` en declaraciones SQL estándar y en estos casos no obtendrás ningún beneficio.

Manejar errores

Cuando se realizan operaciones de bases de datos con PDO, pueden ocurrir errores durante la fase de conexión o durante la ejecución de la consulta. PDO ofrece herramientas robustas para manejar ambos tipos de errores.

Errores de conexión

Si PDO no puede conectarse con la base de datos especificada utilizando el DSN, nombre de usuario y contraseña proporcionados, automáticamente generará una PDOException. Esta excepción puede capturarse utilizando el mecanismo estándar de manejo de excepciones de PHP (abordado con detalle en el capítulo 10), y es posible desplegar un mensaje de error explicando las causas del mismo en el objeto excepción.

Errores de ejecución de la consulta

Si ocurre un error durante la preparación o ejecución de la consulta, PDO proporciona información sobre el mismo con el método `errorInfo()`. Regresa una matriz que contiene tres elementos: el código de error SQL, el código de error de la base de datos y un mensaje de error legible para los humanos (también generado por la base de datos en uso). Es fácil procesar esta matriz y presentar los elementos apropiados que contiene desde la sección de manejo de errores de tu script.

Prueba esto 7-4 Construir un formulario de inicio de sesión

Pongamos ahora en práctica PDO con otra aplicación práctica, una que encontrarás una y otra vez durante tu desarrollo como programador PHP: un formulario de ingreso. Esta aplicación generará un formulario Web para que los usuarios ingresen su nombre de usuario y contraseña; luego verificará estos datos de entrada contra los almacenados en la base de datos y permitirá o rechazará su ingreso. Primero, la aplicación utilizará la base de datos MySQL; sin embargo, después verás qué tan portátil es el código PDO, cuando conmutemos la aplicación hacia una base de datos SQLite.

Utilizar una base de datos MySQL

Para comenzar, arranca el programa cliente de línea de comandos de MySQL y crea una tabla que contenga los nombres de usuario y las contraseñas, como sigue:

```
mysql> CREATE DATABASE app;
Query OK, 0 rows affected (0.07 sec)
mysql> USE app;
Query OK, 0 rows affected (0.07 sec)
mysql> CREATE TABLE usuarios (
    -> id int(4) NOT NULL AUTO_INCREMENT,
    -> nombredeusuario VARCHAR(255) NOT NULL,
    -> contrasena VARCHAR(255) NOT NULL,
    -> PRIMARY KEY (id));
Query OK, 0 rows affected (0.07 sec)
```

En este punto es buena idea "sembrar" la tabla ingresando algunos nombres de usuario y contraseñas. Para simplificarlo, los nombres de usuario serán iguales a las contraseñas, pero cifradas para que estén a salvo de mirones casuales (y de hackers no tan casuales):

```
mysql> INSERT INTO usuarios (nombredeusuario, contrasena)
    -> VALUES ('john', '$1$Tk0.gh4.$42EZDbQ4mOfmXMq.0m1tS1');
Query OK, 1 row affected (0.21 sec)

mysql> INSERT INTO usuarios (nombredeusuario, contrasena)
    -> VALUES ('jane', '$1$.15.tR/.$XK1KW1Wzqy0UuMFKQDHH00');
Query OK, 1 row affected (0.21 sec)
```

Ahora todo lo que se necesita es un formulario de inicio de sesión, y algo de código PHP para leer los valores ingresados en el formulario y compararlos contra los valores almacenados en la base de datos. He aquí el código (*ingreso.php*):

(*continúa*)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Proyecto 7-4: Construir un formulario de ingreso</title>
  </head>
  <body>
    <h2>Proyecto 7-4: Construir un formulario de ingreso</h2>
<?php
    // si el formulario no ha sido enviado
    // genera un nuevo formulario
    if (!isset($_POST['submit'])) {
?>
    <form method="post" action="ingreso.php">
      Nombre de Usuario: <br />
      <input type="text" name="nombredeusuario" />
      <p>
      Contraseña: <br />
      <input type="contraword" name="contrasena" />
      <p>
      <input type="submit" name="submit" value="Ingresar" />
    </form>

<?php
    // si el formulario ha sido enviado
    // verifica los datos proporcionados
    // contra la base de datos
    } else {
      $nombredeusuario = $_POST['nombredeusuario'];
      $contrasena = $_POST['contrasena'];

      // verifica datos de entrada
      if(empty($nombredeusuario)) {
        die('ERROR: Por favor escriba su nombre de usuario');
      }
      if(empty($contrasena)) {
        die('ERROR: Por favor escriba su contraseña');
      }

      // intenta establecer conexión con la base de datos
      try {
         $pdo = new PDO('mysql:dbname=app;host=localhost', 'usuario',
'contra');
      } catch (PDOException $e) {
         die("ERROR: No fue posible conectar: " . $e->getMessage());
      }

      // limpia los caracteres especiales de los datos de entrada
      $nombredeusuario = $pdo->quote($nombredeusuario);
```

```
      // verifica si existe el nombre de usuario
      $sql = "SELECT COUNT(*) FROM usuarios WHERE nombredeusuario =
$nombredeusuario";
      if($result = $pdo->query($sql)) {
        $row = $result->fetch();
        // si es positivo, busca la contraseña cifrada
        if($row[0] == 1) {
          $sql = "SELECT contrasena FROM usuarios WHERE nombredeusuario =
$nombredeusuario";
          // cifra la contraseña ingresada en el formulario
          // la verifica contra la contraseña cifrada que reside en la
base de datos
          // si ambas coinciden, la contraseña es correcta
          if ($result = $pdo->query($sql)) {
            $row = $result->fetch();
            $salt = $row[0];
            if (crypt($contrasena, $salt) == $salt) {
              echo 'Sus credenciales de acceso fueron verificadas
positivamente.';
            } else {
              echo 'Ha ingresado una contraseña incorrecta.';
            }
          } else {
            echo "ERROR: No fue posible ejecutar $sql. " . print_r
($pdo->errorInfo());
          }
        } else {
          echo 'Ha ingresado un nombre de usuario incorrecto.';
        }
      } else {
        echo "ERROR: No fue posible ejecutar $sql. " . print_r
($pdo->errorInfo());
      }

      // cierra conexión
      unset($pdo);
    }
?>
  </body>
</html>
```

La figura 7-12 muestra la apariencia del formulario de ingreso.

(*continúa*)

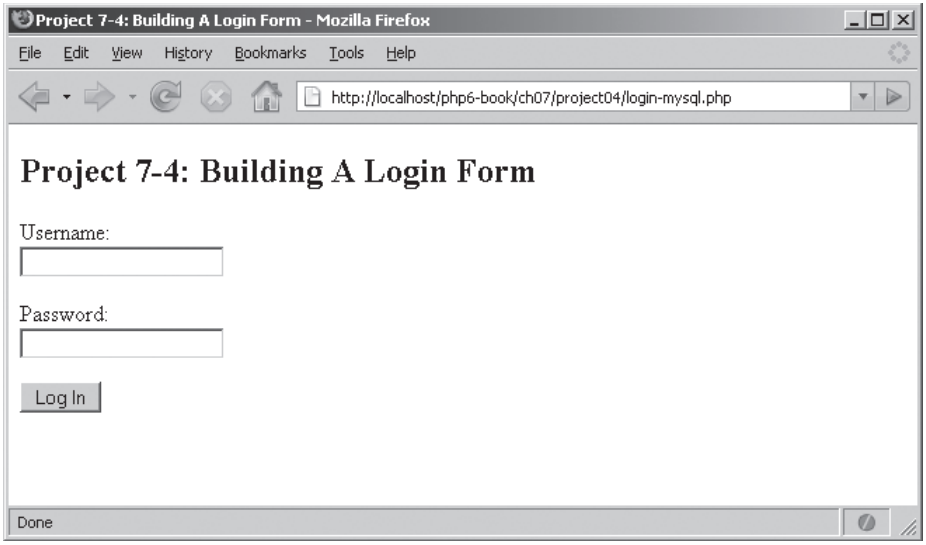

Figura 7-12 Formulario de ingreso en una página Web

Cuando se transmite este formulario, la segunda mitad del script entra en juego. Se realizan varios pasos:

1. Verifica los datos de entrada del formulario para asegurar que el nombre de usuario y la contraseña estén presentes, y detiene la ejecución del script con un mensaje de error si falta cualquiera de los dos. También limpia los caracteres especiales de los datos de entrada utilizando el método `quote()`.
2. Abre una conexión a la base de datos y ejecuta la consulta `SELECT` para verificar si existe una coincidencia con la información almacenada en la base de datos. En caso de que esta verificación dé como resultado un valor falso, genera el mensaje de error "Ha ingresado un nombre de usuario incorrecto".
3. Si el nombre de usuario existe, entonces el script procede a revisar la contraseña. Dado que ésta se cifró utilizando un esquema de cifrado de una sola vía, esta verificación no puede realizarse directamente; la única manera de realizar la tarea es volver a cifrar la contraseña del usuario, a partir de la manera en que ha sido ingresada en el formulario, y comparar esta versión contra la que se encuentra almacenada en la base de datos. En caso de que ambas cadenas de caracteres cifradas coincidan, significa que la contraseña ingresada es correcta.

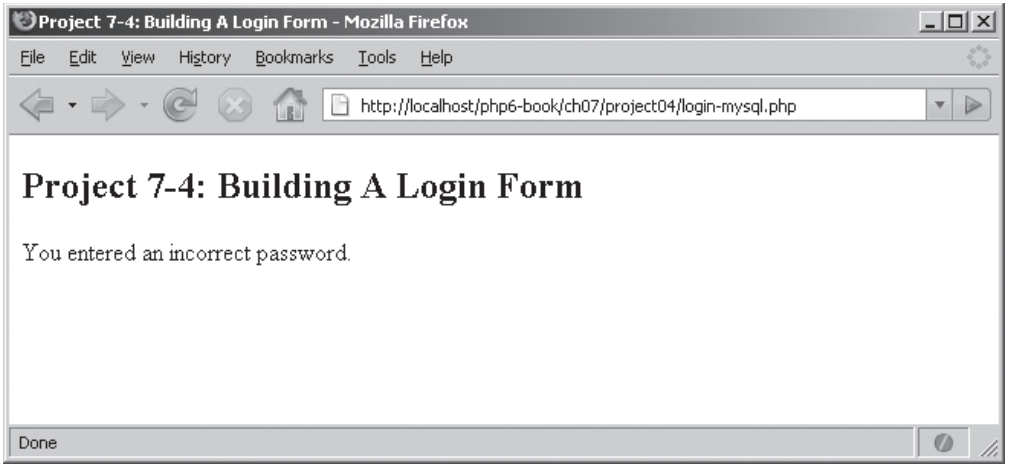

Figura 7-13 El resultado de ingresar una contraseña incorrecta en el formulario de registro

4. Dependiendo del resultado de la prueba, el script genera el mensaje '`Ha ingresado una contraseña incorrecta.`' o bien '`Sus credenciales de acceso fueron verificadas positivamente`'.

La figura 7-13 muestra el resultado de ingresar una contraseña incorrecta y la figura 7-14 muestra el resultado de ingresar correctamente la contraseña.

Figura 7-14 El resultado de ingresar una contraseña válida en el formulario de registro

Pregunta al experto

P: ¿Cómo genero la contraseña cifrada que se utiliza en el ejemplo?

R: Las cadenas de contraseña cifradas que se utilizan en el ejemplo fueron generadas por la función PHP `crypt()`, que utiliza un esquema de cifrado en un solo sentido que aplica a cualquier cadena de caracteres transmitida. El cifrado en un solo sentido hace que la contraseña original sea irrecuperable (de ahí el término "en un solo sentido"). Este cifrado se basa en una clave única o falseada (*salt*), que puede proporcionarse opcionalmente a la función como segundo argumento; en ausencia de ésta, PHP genera de manera automática una clave falseada (*salt*).

La contraseña original ya no es recuperable después de pasar por la función `crypt()`, por lo que realizar después la validación de la contraseña contra el valor proporcionado por el usuario es un proceso de dos pasos: primero, volver a cifrar el valor proporcionado por el usuario con la misma clave falseada utilizada en el proceso original de cifrado y luego comprobar si las dos cadenas cifradas coinciden. Esto es precisamente lo que hace el formulario del ejemplo cuando procesa la información.

Conmutar a una base de datos diferente

Ahora, supongamos que por causas de fuerza mayor te ves forzado a cambiar de MySQL a SQLite. Lo primero que necesitas hacer es crear una tabla en la base de datos para almacenar toda la información de las cuentas de usuario. Así que arranca tu programa cliente SQLite y replica la base de datos creada en la sección anterior (nombra el archivo de base de datos *app.db*):

```
sqlite> CREATE TABLE usuarios (
     -> id INTEGER PRIMARY KEY,
     -> nombredeusuario TEXT NOT NULL,
     -> contrasena TEXT NOT NULL,
     -> );

sqlite> INSERT INTO usuarios (nombredeusuario, contrasena)
     -> VALUES ('john', '$1$Tk0.gh4.$42EZDbQ4mOfmXMq.0m1tS1');
sqlite> INSERT INTO usuarios (nombredeusuario, contrasena)
     -> VALUES ('jane', '$1$.15.tR/.$XK1KW1Wzqy0UuMFQDHH00');
```

El siguiente paso consiste en actualizar el código de tu aplicación para que se comunique con esta base de datos en lugar de hacerlo con la base de MySQL. Como estás utilizando PDO, este cambio consiste en cambiar una (así es, *una*) línea en tu script PHP: el DSN transmitido al constructor del objeto PDO. He aquí el segmento relevante del código:

```
<?php
     // intenta establecer conexión con la base de datos
     try {
```

```
        $pdo = new PDO('sqlite:app.db');
      } catch (PDOException $e) {
         die("Error: No fue posible conectar: " . $e->getMessage());
      }
?>
```

Y ahora, cuando ingreses, tu aplicación trabajará exactamente como lo hizo antes, excepto que ahora utilizará la base de datos SQLite en lugar de MySQL. ¡Inténtalo por ti mismo y lo verás!

Esta portabilidad es precisamente la razón por la que PDO está ganando popularidad entre los desarrolladores de PHP; hace que todo el proceso de conmutar entre bases de datos sea demasiado sencillo, reduciendo tanto el desarrollo y el tiempo de prueba, como los costos asociados a esas tareas.

Resumen

El soporte para bases de datos de PHP es una de las grandes razones de su popularidad, y este capítulo cubrió todo el terreno necesario para que comiences a trabajar con esta importante característica del lenguaje. El capítulo comenzó presentándote una introducción a los conceptos básicos de las bases de datos, y enseñándote las bases del lenguaje estructurado de consultas (SQL). Pasamos rápidamente a presentar las extensiones PHP para bases de datos más populares: las extensiones MySQLi y las extensiones SQLite, además de la extensión de objetos de datos de PHP (PDO). Los temas abordados incluyeron información práctica y ejemplos de código para realizar consultas y modificaciones a las bases de datos con PHP, con el fin de que comiences a utilizar estas extensiones en tu programación diaria.

Para leer más acerca de los temas abordados en este capítulo, te recomiendo que visites los siguientes vínculos:

- Conceptos de bases de datos, en **www.melonfire.com/community/columns/trog/article.php?id=52**
- Bases de SQL, en **www.melonfire.com/community/columns/trog/article.php?id=39** y **www.melonfire.com/community/columns/trog/article.php?id=44**
- Más información sobre fusiones SQL, en **www.melonfire.com/community/columns/trog/article.php?id=148**
- El sitio Web de MySQL, en **www.mysql.com**
- El sitio Web de SQLite, en **www.sqlite.org**
- Extensiones PHP MySQLi, en **www.php.net/mysqli**
- Extensiones PHP SQLite, en **www.php.net/sqlite**
- Extensiones PHP PDO, en **www.php.net/pdo**

Autoexamen Capítulo 7

1. Marca las siguientes declaraciones como verdaderas o falsas:

- **A** Las fusiones de tablas con SQL sólo se realizan entre los campos de llave primaria y llave externa.
- **B** El soporte de PHP para MySQL es reciente.
- **C** La cláusula `ORDER BY` es utilizada para ordenar los campos de una colección de resultados SQL.
- **D** Los campos `PRIMARY KEY` pueden aceptar valores nulos.
- **E** Es posible reescribir el valor generado por SQLite para un campo `INTEGER PRIMARY KEY`.
- **F** Las declaraciones preparadas pueden utilizarse únicamente para las operaciones `INSERT`.

2. Identifica correctamente el comando SQL para cada una de las siguientes operaciones de base de datos:

- **A** Borrar una base de datos.
- **B** Actualizar un registro.
- **C** Borrar un registro.
- **D** Crear una tabla.
- **E** Seleccionar una base de datos para ser utilizada.

3. ¿Qué significa normalizar una base de datos y por qué es útil?

4. Menciona una ventaja y una desventaja de utilizar una biblioteca de abstracción como PDO en lugar de utilizar extensiones nativas de la base de datos.

5. Utilizando la extensión PDO, escribe un script PHP para añadir nuevas canciones a la tabla *canciones* desarrollada en este capítulo. Permite que los usuarios seleccionen el artista de la canción y el rating de una lista de selección desplegable, cuyo contenido sea alimentado por las tablas *artista* y *ratings*.

6. Utilizando la extensión MySQLi de PHP, escribe un script para crear una nueva tabla de base de datos con cuatro campos que tú selecciones. Después, realiza la misma tarea con una base de datos SQLite utilizando la extensión SQLite de PHP.

7. Alimenta manualmente la base de datos MySQL creada en la pregunta anterior con 7 o 10 registros de tu elección. Después, escribe un script basado en PDO que lea el contenido de esta tabla y que migre el contenido encontrado en ella a la tabla de la base de datos SQLite también creada en la pregunta anterior. Utiliza una declaración preparada.