00010111

CHAPTER 23

Relational Database Fundamentals

Learning Objectives

- *State the purpose of a relational database*
- *List and discuss the advantages of a relational database*
- *List and discuss the disadvantages of a relational database*
- *State the purpose of the SQL Transaction Control Language (TCL)*
- *State the purpose of the SQL Data Definition Language (DDL)*
- *State the purpose of the SQL Data Control Language (DCL)*
- *State the purpose of the SQL Data Manipulation Language (DML)*
- *State the purpose of a table*
- *State the purpose of a primary key*
- *State the purpose of a foreign key*
- *State the meaning of a foreign key constraint*
- *State the difference between One-to-Many and Many-to-Many Relationships*
- *Use connection pooling to efficiently connect to a MySQL database*
- *Use prepared statements when executing database queries*
- *Execute database queries with Python*

Introduction

In this chapter you will learn how to use Python to access and interact with a relational database using Structured Query Language (SQL). While many different types of specialized databases exist today, like NoSQL, Graph, Time Series, and Vector, the relational database is the most popular. It's like a draft horse, a Clydesdale, four-wheel drive. You can do a lot with a relational database. And relational databases come in all shapes and sizes, from small footprint implementations that can be embedded within an application, to large-scale, enterprise database clusters with automated replication and multiple read replicas. Pick your poison.

Of all the topics covered in this book, learning even just a little bit of relational database theory, design, and programming gives you an immediately-useful set of tools you can use to design and build complex, data-driven applications.

Theory-wise, you'll learn a handful of new terms including *database*, *table*, *primary key*, and *foreign key*.

On the practical side, you'll need to know the purpose and use of the four SQL language subsets: Data Definition Language (DDL), Data Control Language (DCL), Data Manipulation Language (DML), and Transaction Control Language (TCL). You'll mostly use DML to insert, select, update, and delete data using INSERT, SELECT, UPDATE, and DELETE statements.

You'll also need to understand that while SQL is a standard, there are differences between each vendor's implementation of the language. For example, an SQL command that works on MySQL may not work on Oracle or Microsoft SQL Server.

Design-wise, you need to know how to model your data as a set of one or more tables. You'll need to know how to create one-to-many relationships between tables as well as many-to-many relationships.

Python-wise, you'll need to know how to create a database connection, formulate and execute database queries, and how to process the results.

The theory and fundamental concepts you learn here can be applied to any relational database, however, the actual SQL you write to create or query a MySQL database will not be portable across different database management systems. Also, SQL code that works on an older version of MySQL may not work on a newer version without a few minor changes. This holds true for just about every database management system. Just something to keep in mind.

One more thing. This chapter is decidedly not a complete reference to SQL or relational database theory and design. I have listed several excellent sources in the References section should you wish to pursue these topics further.

OK, let's get crackin'!

1 First Things First

To connect to a MySQL database with Python you'll need to install the `mysql-connector-python` package. This package is available from: *https://pypi.org/project/mysql-connector-python* Before proceeding, however, I'm going to create the project I'll be using to demonstrate the Python code discussed in this chapter. But wait! What am I going to build?

1.1 What I'm Going To Build

To demonstrate the concepts in this chapter, I'll be designing and building a console-based employee training tracker application. I'll use a multilayered application architecture as shown in figure 23-1.

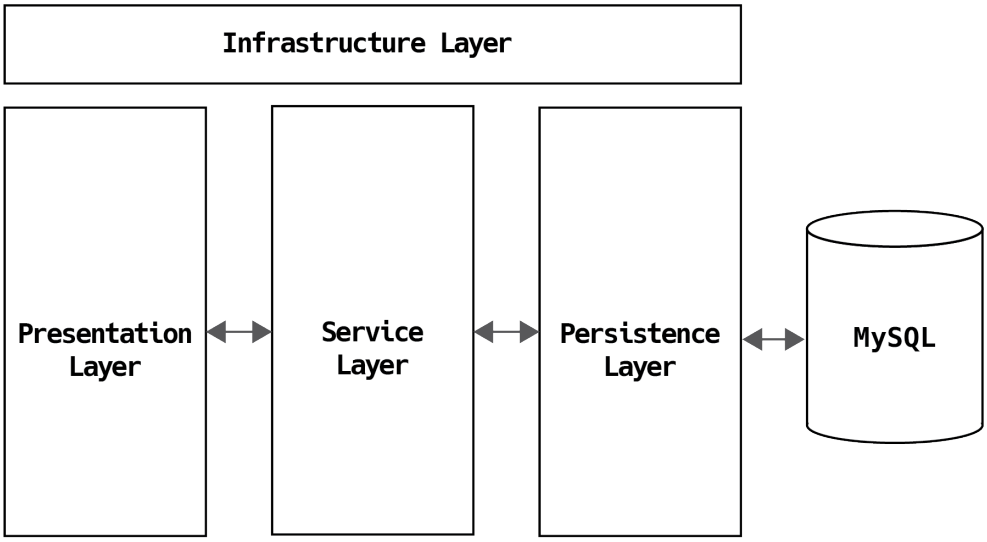

Figure 23-1: Multilayered Application Architecture

Referring to figure 23-1 — A multilayered application architecture assigns specific responsibilities to each layer. Starting from the right, the MySQL database serves as the application's data store. The Persistence Layer is responsible for interacting directly with the database and providing data to the Service Layer. The Service Layer interacts with the Persistence Layer and provides a set of services including data to the Presentation Layer. The Presentation Layer is responsible for providing the application's user interface. The Infrastructure Layer provides objects common to all the other layers.

Note that each layer may contain as many Python modules as required to implement the application. Also, these layers could be deployed together on the same machine or they could be deployed separately on different servers. For example, the database could run locally or on a remote host. The Service Layer might consist of REST endpoints deployed on a separate server, and the Presentation Layer may be a stand-alone client application or web-based interface.

At the very least, organizing your application architecture into separate and distinct layers with specific, well-defined responsibilities provides a logical ordering that makes it easy to reason about how the application code works and where to add features and components as requirements evolve.

The rationale for organizing your application architectures in this fashion is discussed in detail in *Chapter 12: Modules and Functions "Choosing Appropriate Module Names" on page 352*.

1.2 Project Structure

Figure 23-2 shows the project structure I'll be using that reflects the application layers discussed above.

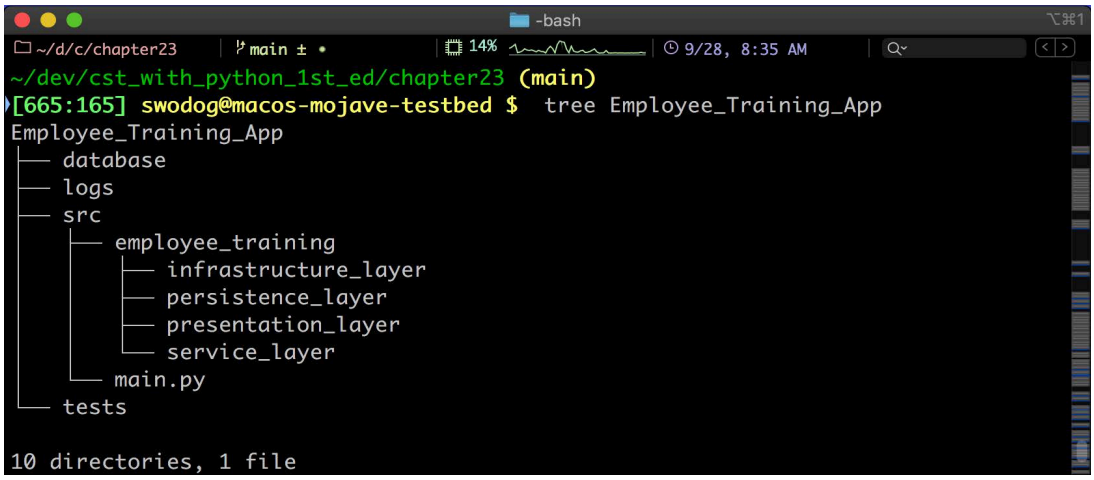

Figure 23-2: Project Structure that Reflects Multiple Application Architectural Layers

Referring to figure 23-2 — I have named the project folder `Employee_Training_App` and added four subdirectories: `database`, `logs`, `src`, and `tests`. Within the `src` directory I have added a directory named `employee_training` which contains the following subdirectories: `infrastructure_layer`, `persistence_layer`, `presentation_layer`, and `service_layer`. I have also added a `main.py` file which will serve as the application entry point. For guidance on how to organize your projects see *Chapter 9: Project Organization "Baseline Project Structure" on page 288*.

1.3 Don't Panic!

Alright, don't panic. I'm not going to build the entire application in this chapter. I will use it to demonstrate relational database concepts while focusing primarily on the Persistence Layer. I will then extend the application in *Chapter 24: Scripting The Database* and *Chapter 25: Unit Testing*.

1.4 Install the mysql-connector-python Package

To install the `mysql-connector-python` package, first, navigate to the `Employee_Training_App` directory and create a virtual environment with `pipenv` like so:

```
pipenv --python 3.12
```

The results of running this command should look similar to what's shown in figure 23-3.

Referring to figure 23-3 — When the `pipenv` command completes, you should see a `.venv` subdirectory added to your project. You can now install the `mysql-connector-python` package with the following command:

```
pipenv install mysql-connector-python
```

The results of running this command should look similar to that shown in figure 23-4.

Referring to figure 23-4 — If the command completes successfully, you are cleared to proceed with the rest of the chapter. You'll need to pay attention to how I run the Python examples because I'm using a virtual environment.

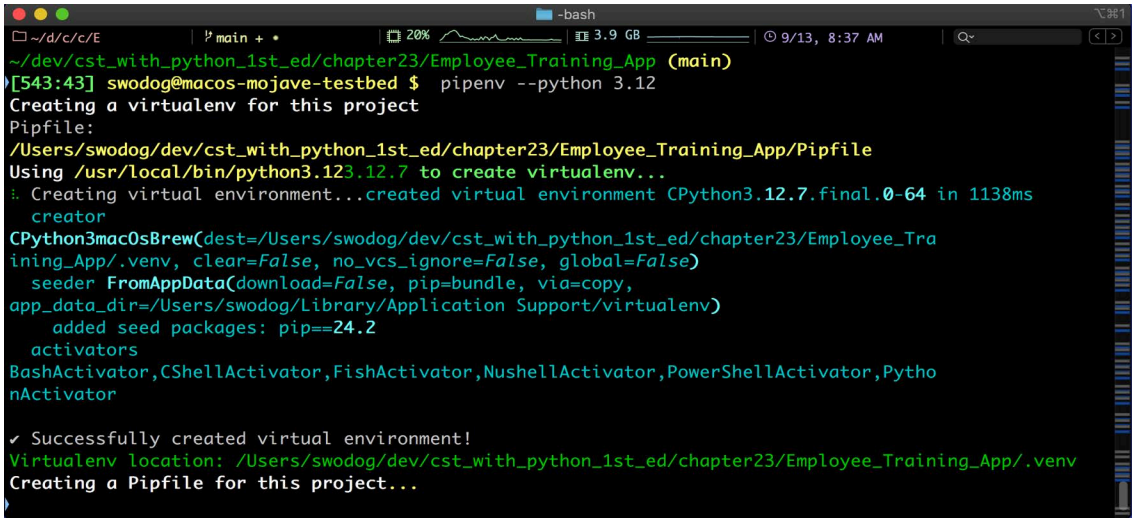

Figure 23-3: Results of Creating Virtual Environment with `pipenv`

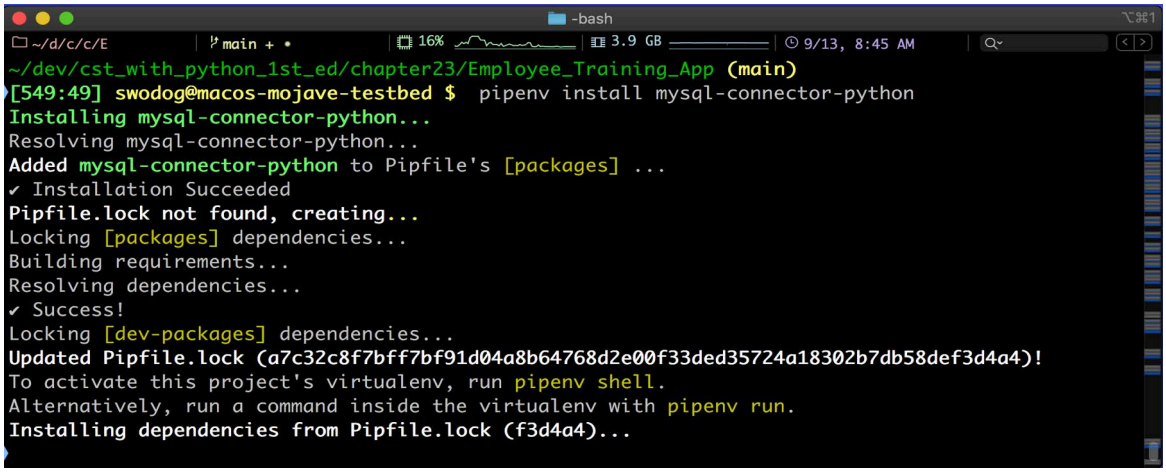

Figure 23-4: Installing the `mysql-connector-python` Package with `pipenv`

Quick Review

A multilayered application architecture assigns separate and distinct responsibilities to each application layer. The Employee Training Application's architecture includes the Persistence Layer, Service Layer, Presentation Layer, and the Infrastructure Layer.

Application layers provide a convenient and logical way to organize your project. This goes a long way towards managing physical and conceptual complexity.

To connect to a MySQL database with Python use the `mysql-connector-python` package available from PyPI.

2 Relational Database concepts

A *database management system* (DBMS) is a software application that stores data in some form or another, usually on the file system, and provides a suite of software tools that enables

users to create, manipulate, and delete database objects and data. The term *database* refers to a related collection of data. A DBMS may contain one or more databases. The term database is often used interchangeably to refer both to a DBMS and to the databases it contains. For example, a colleague is more likely to ask you "What type of database are you going to use?" rather than "What type of database management system are you going to use?"

A *relational database* stores data in relations referred to as *tables*. A *relational database management system* (RDBMS) is a software application that allows users to create and manipulate relational databases. Popular RDBMS systems I'm personally familiar with include Oracle, MySQL, PostgreSQL, and Microsoft SQL Server, but there exist many more.

A *table* is composed of *rows* and *columns*. Each column has a *name* and an associated database *type*. For example, a table named `employees` may have a column named `first_name` with a type of `varchar(50)`. (*i.e.,* A variable-length character field with a 50 character limit.) Each row is an instance of data stored in the table. For example, the `employees` table might contain any number of employee entries, with each entry occupying a single row in the table.

In most cases it is desirable to be able to uniquely identify each row of data contained within a table. To do this, one or more of the table columns must be designated as the *primary key* for that table. The important characteristic of a primary key is that its value must be unique for each row.

The power of relational databases derives from their ability to define associations between different tables. One table can be related to another table by the addition of a *foreign key*. The *primary key* of a *parent* table serves as the *foreign key* in the related, or *child*, table, as figure 23-5 illustrates.

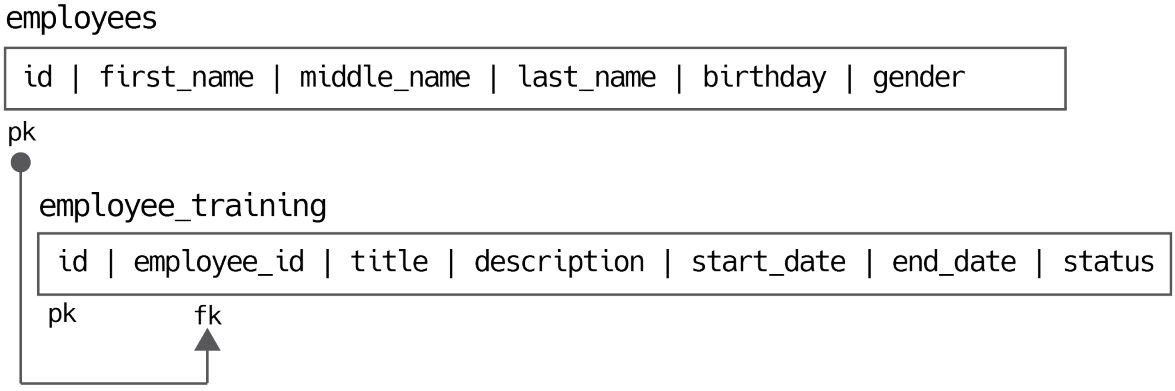

Figure 23-5: The Primary Key of One Table Can Server as the Foreign Key in a Related Table

Referring to figure 23-5 — The `id` column serves as the primary key for the `employees` table. An `employee_id` column in the `employee_training` table serves as a foreign key for that table. In this manner, a relationship has been established between the `employees` table and the `employee_training` table. These tables can now be manipulated together to extract meaningful data regarding employees and the training they have taken. A table can be related to multiple tables by the inclusion of multiple foreign keys.

Primary keys and foreign keys can be used together to enforce *referential integrity*. For example, you should not be able to insert a new row into the `employee_training` table unless the `employee_id` foreign key value you are trying to insert already exists as a primary key in the `employees` table. Also, what should happen when an employee row is deleted from the `employees` table? A *cascade delete* foreign key constraint can automatically delete any related records in

the `employee_training` table. When an employee row is deleted from the `employees` table, any rows in the `employee_training` table with a matching foreign key will also be deleted.

2.1 Structured Query Language (SQL)

SQL is used to create, manipulate, and delete relational database objects and the data they contain. Although SQL is a standardized database language, each RDBMS vendor is free to add extensions to the language, which essentially renders the language non-portable between different database products. What this means to you is that while the examples I present in this chapter will work with MySQL, they may not work with Oracle, Microsoft SQL Server, or whatever relational database system you're familiar with. This holds true especially for the SQL Data Definition Language commands, which we will cover shortly.

SQL comprises four sub-languages, which group commands according to functionality: *Data Definition Language (DDL)*, *Data Manipulation Language (DML)*, *Data Control Language (DCL)*, which is used to grant or revoke user rights and privileges on database objects, and *Transaction Control Language (TCL)*, which is used to manage transactions. A transaction is a series of database commands that must all execute to completion before the operation can be considered complete and *committed*, or, in case of failure, the commands must all be *rolled back*. In this section I will focus on the use of DDL and DML.

2.2 Data Definition Language (DDL)

The DDL includes the `CREATE`, `USE`, `ALTER`, and `DROP` commands. Let's use a few of these commands to set up the employee training database that will be used to store data for the employee training application. Before we begin, launch phpMyAdmin and take a look at the default databases provided upon installation as shown in figure 23-6.

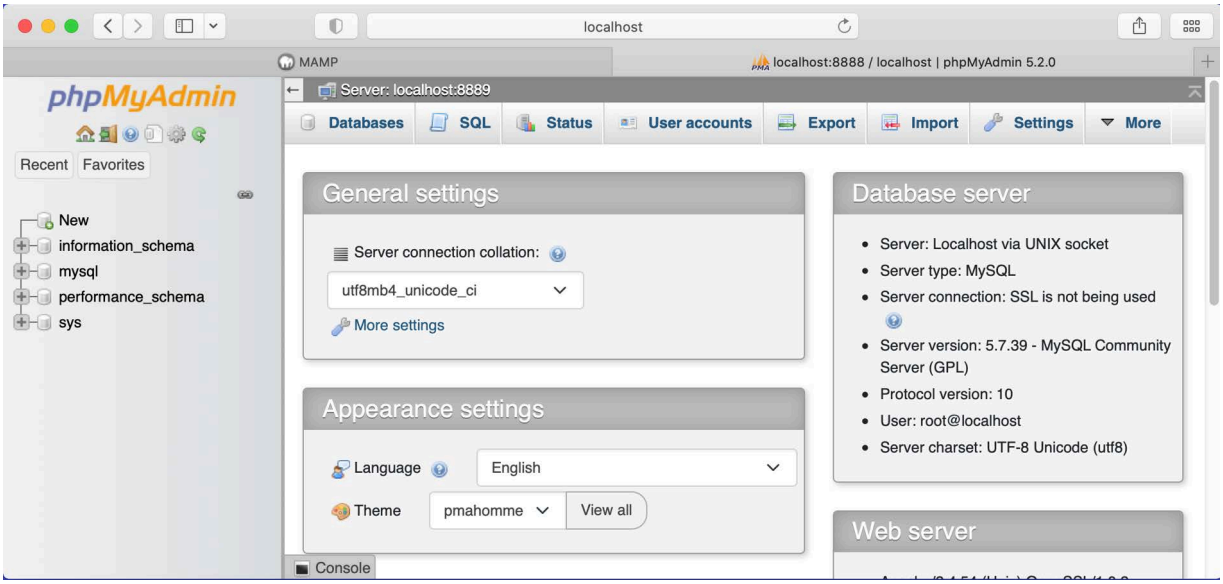

Figure 23-6: MySQL Server's Default Databases

Referring to figure 23-6 — MySQL server initially deploys with the four databases listed in the left-hand column: *information_schema*, *mysql*, *performace_schema*, and *sys*. (On Linux there's a *phpmyadmin* database as well.) Generally speaking, you don't want nor usually need to

fiddle with these databases because making an unintended change can break your MySQL Server instance. I do, however, recommend exploring their contents. To learn more about the purpose and contents of these databases see *https://dev.mysql.com/doc/refman/8.4/en/system-schema.html*.

2.2.1 Creating the `employee_training` Database

You could create the database using phpMyAdmin, but first let's learn how to use these commands from the console. Launch a terminal and run `mysql`. At the `mysql>` prompt, type the following command:

```
CREATE DATABASE employee_training;
```

If this command completes successfully you should see the following output:

```
Query OK, 1 row affected (0.01 sec)
```

To verify the database exists, you can either refresh the phpMyAdmin page or type the following command at the `mysql>` prompt:

```
SHOW databases;
```

Figure 23-7 shows the results of running these commands.

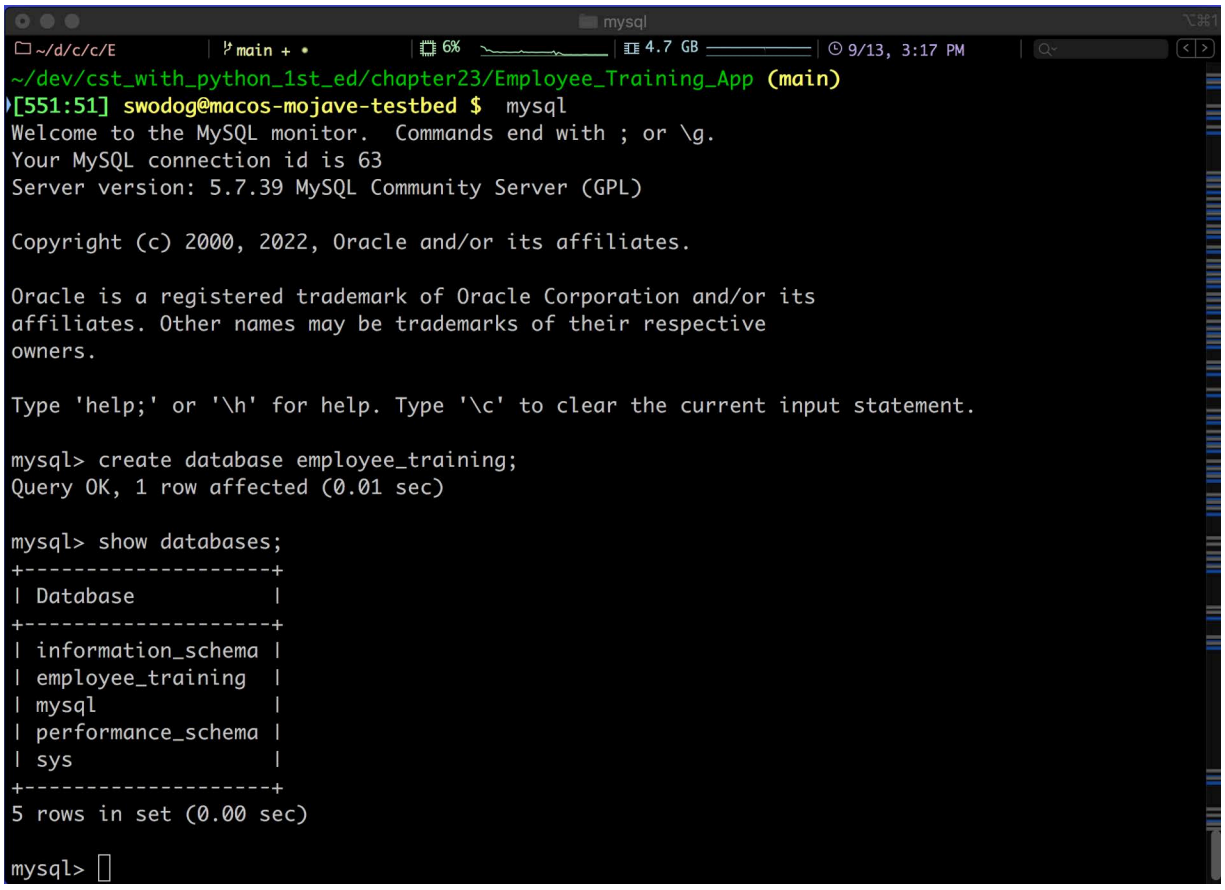

Figure 23-7: Creating the `employee_training` Database

2.2.2 Creating a Database with a Script

Let's automate the database creation command. Navigate to the `Employee_Training_App` directory, launch Visual Studio Code, and create the SQL script shown in example 23.1.

23.1 create_database.sql

```
1   /* ****************************************
2      Create the employee_training database.
3      Drop the database if it already exists.
4   **************************************** */
5
6   DROP DATABASE IF EXISTS `employee_training`;
7   CREATE DATABASE IF NOT EXISTS `employee_training`;
8
```

Referring to example 23.1 — Save this script in the `database` directory. Lines 1 through 4 contain a multiline comment. These are the same types of multiline comments used in programming languages like C, C++, C#, and Java. On line 6, I added a DROP DATABASE command to drop the database if it already exists. This clears the way for line 7 to execute.

To run this script, launch a terminal, navigate to the `database` directory, and enter the following command:

```
mysql < create_database.sql
```

If all goes well, you'll see no output as a result of running this command. To verify the database exists, start the `mysql` client and at the `mysql>` prompt type the following command:

```
SHOW databases;
```

You should see the `employee_training` database listed in the output. OK, now that you know how to drop and create a database with a script, let's create a table.

2.2.3 Creating Tables

You *could* create a table from the `mysql>` prompt, but doing so is tedious and error-prone. It's best to use a script, that way, when you dork things up, and you **will** dork things up, you can simply fix the problem and rerun the script. Example 23.2 gives the SQL script to create the `employees` table shown earlier in figure 23-5.

23.2 create_tables.sql

```
1   /* ************************************************************
2      Drop and Create the tables for the employee_training database.
3   ************************************************************ */
4
5   -- Switch to employee_training database
6   USE`employee_training`
7
8   -- Drop the table if it exists
9   DROP TABLE IF EXISTS `employees`;
10
11  -- Create the table
12  CREATE TABLE IF NOT EXISTS `employees` (
13      `id` int(11) NOT NULL,
14      `first_name` varchar(25) NOT NULL,
15      `middle_name` varchar(25) NOT NULL,
16      `last_name` varchar(25) NOT NULL,
17      `birthday` varchar(25) NOT NULL,
18      `gender` char(1) NOT NULL
19  );
```

```
20
21   -- Designate the `id` column as the primary key
22   ALTER TABLE `employees`
23     ADD PRIMARY KEY (`id`);
24
25   -- Make `id` column auto increment on inserts
26   ALTER TABLE `employees`
27     MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;
28
```

Referring to example 23.2 — From the top, lines 1 through 3 contain a multiline comment. You saw those earlier. Line 5 contains a single-line comment which starts with two hyphens "--". The `USE` statement on line 6 selects the `employee_training` database. This is **very important** and leads me to my first Pro Tip.

PRO TIP: Make sure you are in the correct database before running database scripts. Switch to the target database with the `USE` statement.

On line 9, I call `DROP TABLE` to drop the `employees` table if it exists, and call `CREATE TABLE`, which begins on line 12, if it does not exist. On line 22, I call `ALTER TABLE` to designate the `id` column as the table's primary key. Finally, on line 26, I call `ALTER TABLE` again to modify the `id` column to make it `AUTO_INCREMENT`. This will automatically handle the insertion of a unique integer primary key value into the `employees` table's `id` column during an `INSERT`.

To run this script, navigate to the project's `database` directory and type the following command:

```
mysql < create_tables.sql
```

To verify the table exists, launch phpMyAdmin, select the `employee_training` database and expand the `employees` table columns as shown in figure 23-8.

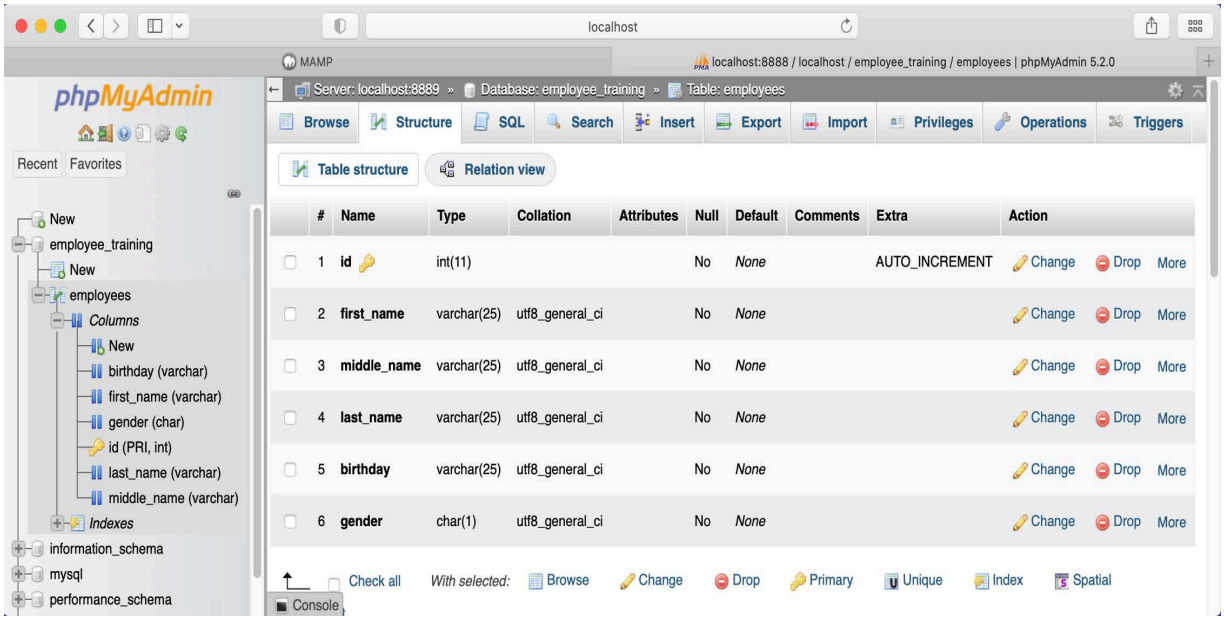

Figure 23-8: Verify `employees` Table Successfully Created

Referring to figure 23-8 — I have also selected the **Structure** tab which shows more detail about the `employees` table column attributes. You can also get this information via the `mysql` client with the help of the `DESCRIBE` command as shown in figure 23-9.

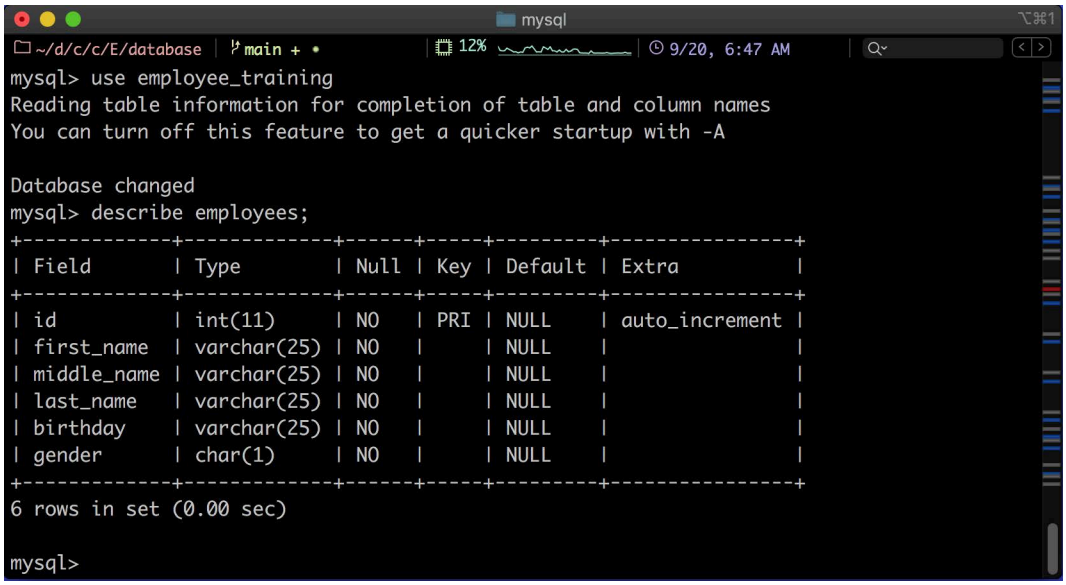

Figure 23-9: Using DESCRIBE to List Table Structure

Referring to figure 23-9 — Notice at the top I used the `USE employee_training` command to switch to the `employee_training` database before using the `DESCRIBE` command.

2.2.4 SQL Data Types

Again referring to example 23.2 and figure 23-9 — Database columns must be assigned a type. I have used three different types in the `employees` table: `INT`, `CHAR`, and `VARCHAR`. MySQL supports four primary categories of data types: *Numeric*, *Sting*, *Spatial*, and *JSON*. To dive deeper into MySQL data types see *https://dev.mysql.com/doc/refman/8.4/en/data-types.html*.

2.2.5 Taking Stock

OK, you've seen the `DROP`, `CREATE`, `USE`, and `ALTER` statements in action. You also have two database scripts, `create_database.sql` and `create_tables.sql`, which you can use to quickly drop and create the `employee_training` database and the `employees` table. That's about as deep into the SQL Data Definition Language (DDL) I'll go in this chapter. What we need now is to insert some test data into the `employees` table so we can run some queries.

2.3 Data Manipulation Language (DML)

MySQL supports the following four primary Data Manipulation Language (DML) commands: `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

2.3.1 INSERT Command

Let's start with the INSERT command and create an SQL script that inserts some test data into the `employees` table as shown in example 23.3.

23.3 insert_test_data.sql

```
1    /* ******************************************************
2       Insert test data into the employee_training database.
3    ******************************************************/
4
5    -- Switch to the employee_training database.
6    USE `employee_training`
7
8    -- Insert data into the employees table.
9    INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
10          VALUES ('Rick', 'Warren', 'Miller', '1976-08-20 00:00:00', 'M');
11
```

Referring to example 23.3 — I'm only inserting one record into the employees table to make sure it works. Notice on line 6, I switch to the `employee_training` database with the help of the USE command. On line 9, I use the INSERT command to insert one row of data into the `employees` table. The column names you want to insert data into appear within the parentheses on line 9. The VALUES you want to go into each column appear within the parentheses on line 10 in the same order as the column list on line 9. **Don't forget the semicolon at the end**! That's MySQL's signal to execute the statement. This leads me to my next Pro Tip:

PRO TIP: Don't forget the semicolon at the end of SQL statements.

OK, to run this script, launch a terminal, navigate to the project's `database` directory and type the following command:

```
mysql < insert_test_data.sql
```

If all goes well, you won't see any output. Let's verify the data with a query.

2.3.2 SELECT Command

To verify the data was inserted you can browse the table in phpMyAdmin or launch the `mysql` client and type the following commands at the `mysql>` prompt:

```
USE employee_training
SELECT * from employees;
```

In this case, the `'*'` means select *all* the columns from the `employees` table. Your output should look similar to that shown in figure 23-10.

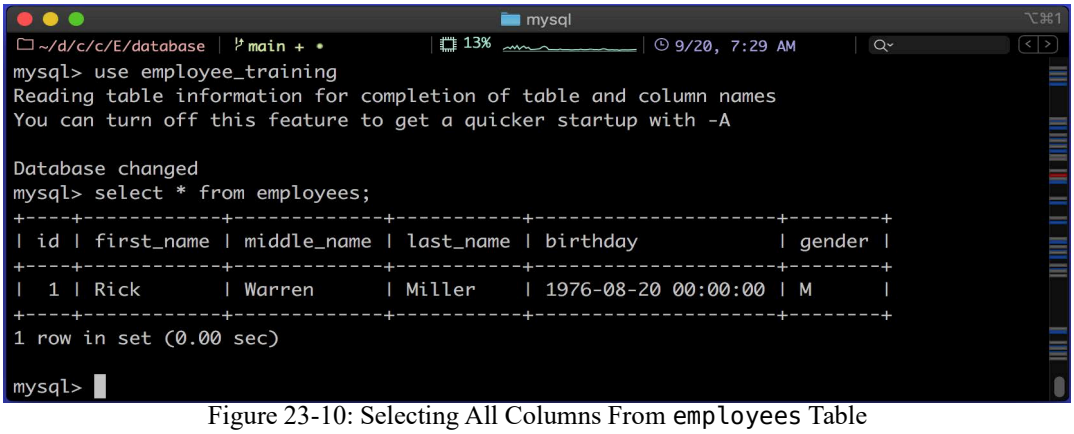

Figure 23-10: Selecting All Columns From `employees` Table

Referring to figure 23-10 — The database isn't very exciting with only one record. Let's add several more employees and look at a few more Data Manipulation Language commands. Example 23.4 gives the expanded `insert_test_data.sql` script.

23.4 insert_test_data.sql

```
/* ******************************************************
   Insert test data into the employee_training database.
****************************************************** */

-- Switch to the employee_training database.
USE `employee_training`

-- Insert data into the employees table.
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Rick', 'Warren', 'Miller', '1976-08-20 00:00:00', 'M');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Jose', 'Miguel', 'Pi', '1985-12-01 00:00:00', 'M');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Coralie', 'Sarah', 'Miller', '1989-04-28 00:00:00', 'F');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Melissa', 'Martin', 'Miller', '1992-06-11 00:00:00', 'M');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Kyle', 'Victor', 'Miller', '1990-08-15 00:00:00', 'M');

```

Referring to example 23.4 — This gives me a few more records to work with. Feel free to modify the script to suit your needs. To run this script, you'll need to run the `create_tables.sql` script first to drop and recreate the employees table. You could also run all three SQL scripts in the following order: `create_database.sql`, `create_tables.sql`, `insert_test_data.sql`. To do this, launch a terminal, navigate to the project's `database` directory, and run the following commands:

```
mysql < create_database.sql
mysql < create_tables.sql
mysql < insert_test_data.sql
```

When these commands complete successfully, launch the `mysql` client, and at the `mysql>` prompt run the following commands:

```
USE employee_training
SELECT first_name, middle_name, last_name, birthday, gender FROM employees;
```

Figure 23-11 shows the results of running these commands.

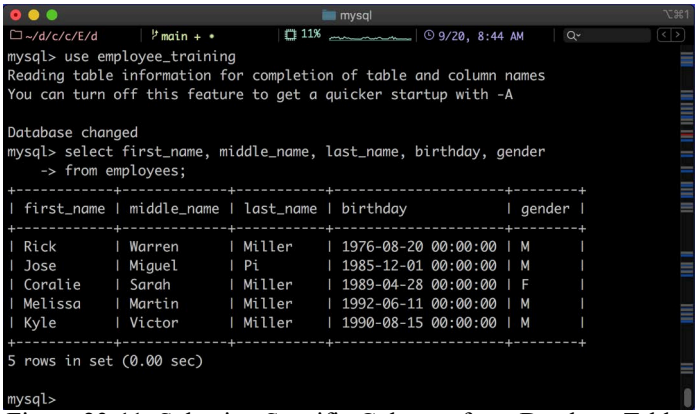

Figure 23-11: Selecting Specific Columns from Database Table

Referring to figure 23-11 — In this example, I am selecting specific column names. As you'll learn later when we query the database with Python, you will want to select columns by name in a specific order because doing so enables you to add columns to a database table and not break existing queries. This will make more sense when you see an example. Also notice that in figure 23-11 that I have broken the `SELECT` statement over multiple lines.

OK, now that we have more test data, let's start asking the database some questions. After all, what good is a database and its data if it can't answer questions? A few of the questions this simple database can answer include:

How many employees do we have?

```
SELECT COUNT(*) FROM employees;
```

How many employees have a last name of Miller?

```
SELECT COUNT(*) FROM employees WHERE last_name = 'Miller';
```

Figure 23-12 shows the results of running these commands.

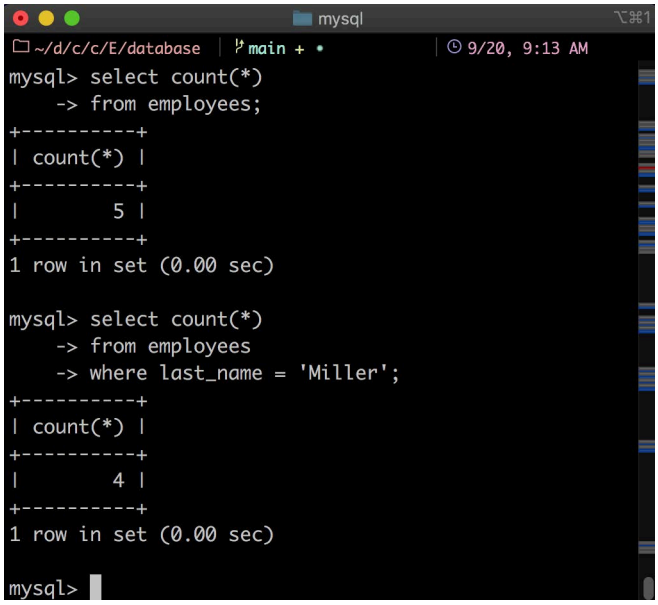

Figure 23-12: Asking The Database Questions

Referring to figure 23-12 — Notice how I structure the SELECT statements and place the FROM and WHERE clauses on separate lines like so:

```
SELECT
FROM
WHERE
```

This structure goes a long way towards making your SELECT statements easier to read and debug. Also note that the term *query* is synonymous with the notion of *asking the database a question*.

2.3.3 UPDATE Command

As its name implies, the UPDATE command is used to make changes to existing data. To demonstrate its use, suppose all the employees with a last name of Miller decided to change their last name to Johnson. You could make that change in the database with the following command:

```
UPDATE employees SET last_name = 'Johnson' WHERE last_name = 'Miller';
```

The output from this command will look something like this:

```
Query OK, 4 rows affected (0.01 sec)
Rows matched: 4  Changed: 4  Warnings: 0
```

To verify the update use the following query:

```
SELECT first_name, last_name FROM employees;
```

Figure 23-13 shows the results of running these commands.

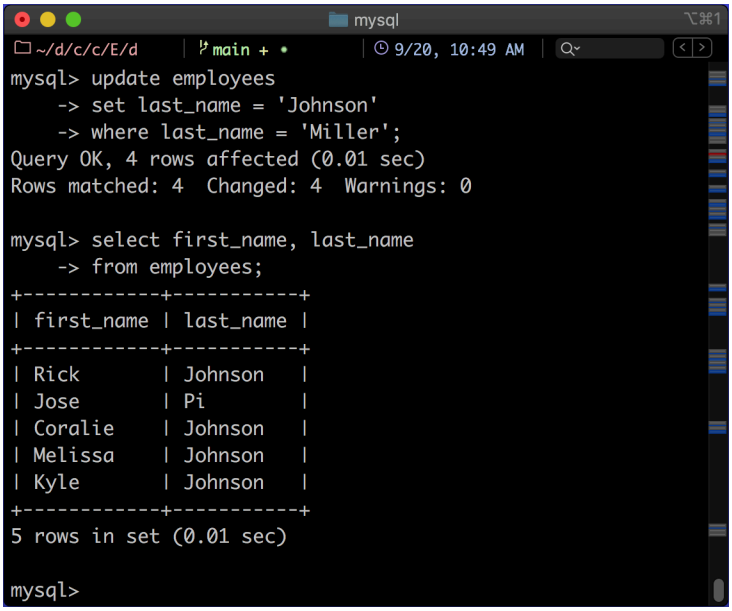

Figure 23-13: Using UPDATE to Make Changes to Existing Data

Referring to figure 23-13 — The `UPDATE` command returns the number of affected rows. The `WHERE` clause is used to specify which records to apply the update to. In this case the update is applied to all records whose `last_name` column equals the string `'Miller'`.

2.3.4 DELETE Command

The DELETE command is used to delete existing rows from a database table. To demonstrate its use, I'll delete all employees whose first names start with the letter 'R'. The following command should do the trick:

```
DELETE FROM employees WHERE LEFT(first_name, 1) = 'R';
```

To verify the delete, query the `employees` table:

```
SELECT first_name, last_name FROM employees;
```

Figure 23-14 shows the results of running these commands.

Referring to figure 23-14 — The `LEFT()` function returns the first letter of the `last_name` column. Earlier I used the `COUNT()` function to count the number of rows in the employees table. For a complete list of MySQL functions see *https://dev.mysql.com/doc/refman/8.4/en/functions.html*. You can see from the results of the SELECT statement the row with the `first_name` column `'Rick'` was deleted.

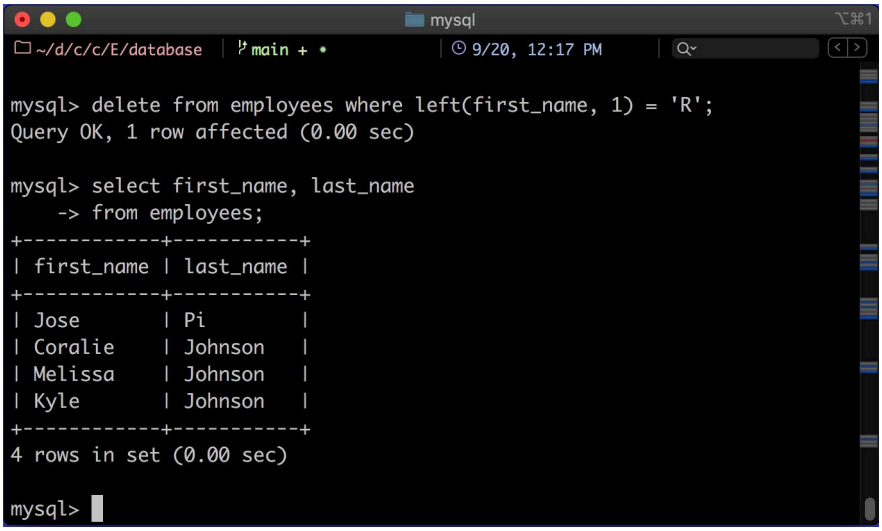

Figure 23-14: Deleting All `employee` Rows whose `first_name` Starts with the Letter `'R'`

2.4 Parting Thoughts About Data Manipulation Language

In this section, I focused on the DML statements INSERT, SELECT, UPDATE, and DELETE, along with two functions COUNT() and LEFT(). For the complete list of MySQL DML statements see *https://dev.mysql.com/doc/refman/8.4/en/sql-data-manipulation-statements.html*.

Quick Review

A database management system (DBMS) is a software application that stores data in some form or another, usually on the file system, and provides a suite of software components that enables users to create, manipulate, and delete database objects and data. The term database refers to a related collection of data.

A relational database stores data in relations referred to as tables. A relational database management system (RDBMS) is a software application that allows users to create and manipulate relational databases.

A table is composed of rows and columns. Each column has a name and an associated database type. In most cases it is desirable to be able to uniquely identify each row of data contained within a table. To do this, one or more of the table columns must be designated as the primary key for that table. The important characteristic of a primary key is that its value must be unique for each row.

The power of relational databases derives from their ability to define associations between different tables. One table can be related to another table by the implementation of a foreign key. The primary key of a parent table serves as the foreign key in the related, or child, table. Primary keys and foreign keys can be used together to enforce referential integrity.

Structured Query Language is used to create, manipulate, and delete relational database objects and the data they contain. Although SQL is a standardized database language, each RDBMS vendor is free to add extensions to the language, which essentially renders the language non-portable between different database products.

SQL comprises four sub-languages, which group commands according to functionality: Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), which is used to grant or revoke user rights and privileges on database objects, and Transaction Control Language (TCL), which is used to manage transactions. A transaction is a series of database commands that must all execute to completion before the operation can be considered complete, or, in case of failure, the commands must all be rolled back.

Data Definition Language (DDL) includes the `CREATE`, `USE`, `ALTER`, and `DROP` commands. Data Manipulation Language (DML) includes the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` commands.

3 Related Tables

The power of relational databases lies in their ability to define relationships between tables. The two most common types of relationships include the one-to-many and the many-to-many. Let's start with the one-to-many relationship.

3.1 The One-to-Many Relationship

Let's now add the `employee_training` table to the database. Example 23.5 shows the modified `create_tables.sql` script.

23.5 create_tables.sql

```
1    /* ************************************************************
2       Drop and Create the tables for the employee_training database.
3    ************************************************************ */
4
5    -- Switch to employee_training database
6    USE `employee_training`
7
8    -- Drop the table if it exists
9    DROP TABLE IF EXISTS `employees`;
10
11   -- Create the table
12   CREATE TABLE IF NOT EXISTS `employees` (
13       `id` int(11) NOT NULL,
14       `first_name` varchar(25) NOT NULL,
15       `middle_name` varchar(25) NOT NULL,
16       `last_name` varchar(25) NOT NULL,
17       `birthday` varchar(25) NOT NULL,
18       `gender` char(1) NOT NULL
19   );
20
21   -- Designate the `id` column as the primary key
22   ALTER TABLE `employees`
23     ADD PRIMARY KEY (`id`);
24
25   -- Make `id` column auto increment on inserts
26   ALTER TABLE `employees`
27     MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;
28
29   -- Drop employee_training table if it exists
30   DROP TABLE IF EXISTS `employee_training`;
31
```

```
32  -- Create employee_training table
33  CREATE TABLE `employee_training` (
34      `id` int(11) NOT NULL,
35      `employee_id` int(11) NOT NULL,
36      `title` varchar(100) NOT NULL,
37      `description` varchar(250) NOT NULL,
38      `start_date` varchar(25) NOT NULL,
39      `end_date` varchar(25) NOT NULL,
40      `status` varchar(25) NOT NULL
41  );
42
43  -- Make employee_training.id column primary key
44  -- And create an index on the employee_id column
45  ALTER TABLE `employee_training`
46    ADD PRIMARY KEY (`id`),
47    ADD KEY `employee_training_ibfk_1` (`employee_id`);
48
49  -- Make employee_training.id column Auto Increment
50  ALTER TABLE `employee_training`
51    MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;
52
53  -- Add Cascade Delete Constraint
54  ALTER TABLE `employee_training`
55    ADD CONSTRAINT `employee_training_ibfk_1`
56    FOREIGN KEY (`employee_id`) REFERENCES `employees` (`id`)
57    ON DELETE CASCADE
58    ON UPDATE CASCADE;
59
```

Referring to example 23.5 — The action starts on line 30 where I `DROP` the `employee_training` table if it exists. The `CREATE TABLE` statement starts on line 33. The `ALTER TABLE` statement on line 45 marks the `id` column as the `PRIMARY KEY` and adds an index named `employee_training_ibfk_1` on the `employee_id` column. The `ALTER TABLE` statement on line 50 sets the `id` column to `AUTO_INCREMENT`, and finally, the `ALTER TABLE` statement that starts on line 54 adds a *foreign key constraint* that will automatically delete related rows from the `employee_training` table when a parent row from the `employees` table is deleted. The `ON UPDATE CASCADE` will update related primary keys (`employee_id`) if a primary key in the `employees` table is modified.

Now, to run this script, you'll need to first run the `create_database.sql` script followed by the modified `create_tables.sql` script. To do this, launch a terminal, navigate to the project's `database` directory, and enter the following commands:

```
mysql < create_database.sql

mysql < create_tables.sql
```

You can then launch `mysql` and at the `mysql>` prompt verify the new table exists with the DESCRIBE command like so:

```
USE employee_training

DESCRIBE employee_training
```

Figure 23-15 shows the results of running these commands.

Referring to figure 23-15 — The table structure looks fine to me. Notice the output table's Key column. The value `'PRI'` stands for `PRIMARY KEY`, which means the `employee_training` table's `id` column can only contain unique values. The `employee_id` column is tagged with the Key value `'MUL'`, which means it's part of an index that can contain multiple non-unique values.

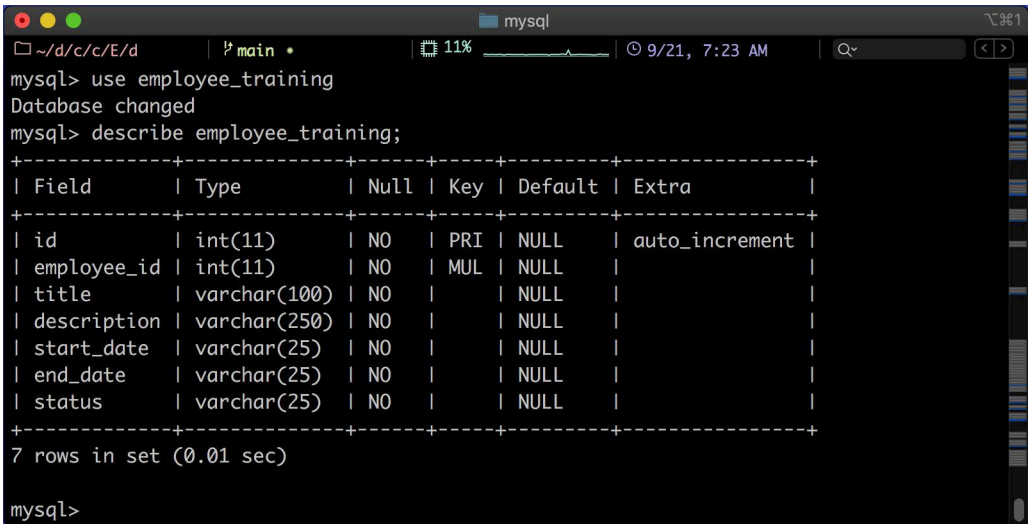

Figure 23-15: Verifying the `employee_training` Table Structure with the DESCRIBE Command

Guess what we need now? If you spit out your coffee and shouted "Test Data!" You're right! Example 23.6 gives the modified `insert_test_data.sql` script.

23.6 insert_test_data.sql

```
/* ******************************************************
   Insert test data into the employee_training database.
****************************************************** */

-- Switch to the employee_training database.
USE `employee_training`

-- Insert data into the employees table.
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Rick', 'Warren', 'Miller', '1976-08-20 00:00:00', 'M');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Jose', 'Miguel', 'Pi', '1985-12-01 00:00:00', 'M');


-- Insert data into the employee_training table.
INSERT INTO `employee_training` (employee_id, title, description, start_date,
                                 end_date, status)
      VALUES(1, 'Intro to Bash Shell Scripting',
             'How to script the Bash shell.',
             '2025-09-15 00:00:00', '2025-09-20 00:00:00', 'Pass');
INSERT INTO `employee_training` (employee_id, title, description, start_date,
                                 end_date, status)
      VALUES(1, 'Intro to C Programming',
             'Introduces student to C programming language fundamentals.',
             '2025-02-03 00:00:00', '2025-02-07 00:00:00', 'Pass');
INSERT INTO `employee_training` (employee_id, title, description, start_date,
                                 end_date, status)
      VALUES(2, 'Managing Difficult Employees',
             'How to lead through kindness.',
             '2025-03-10 00:00:00', '2025-03-10 00:00:00', 'Pass');
INSERT INTO `employee_training` (employee_id, title, description, start_date,
                                 end_date, status)
      VALUES(2, 'Intro to Bash Shell Scripting',
             'How to script the Bash shell.',
```

```
35              '2025-09-15 00:00:00', '2025-09-20 00:00:00', 'Pass');
36    INSERT INTO `employee_training` (employee_id, title, description, start_date,
37                                end_date, status)
38          VALUES(2, 'Database Programming with Python',
39                 'How to develop data-driven apps with Python.',
40                 '2025-06-16 00:00:00', '2025-06-20 00:00:00', 'Pass');
41
```

Referring to example 23.6 — I'm only creating two employees to reduce the size of the script and allow room for training data. The most important concepts you must keep in mind when inserting data into a related table is that the value you insert into the foreign key column, which in this case is the `employee_id` column, must exist as a primary key value in the parent table. That's why I first insert rows into the `employees` table and use the resulting primary keys to set the `` `employee_training`.employee_id `` column values. Now, I just know from experience that if I start from scratch, that is, with an empty database, the primary key value of the first row inserted into an `AUTO_INCREMENT PRIMARY KEY` field will be `1`, the next value will be `2`, and so on and so forth. So, I just use these values when I insert data into the `employee_training` table.

3.1.1 Clearing The Decks

OK, just to ensure you are starting with a clean database, run all three scripts.

```
mysql < create_database.sql
```

```
mysql < create_tables.sql
```

```
mysql < insert_test_data.sql
```

Next, launch the `mysql` client and from the `mysql>` prompt enter the following query:

```
USE employee_training
```

```
SELECT first_name, last_name, title, status
FROM employees, employee_training
WHERE `employees`.id = employee_id;
```

Your results should look similar to that shown in figure 23-16.

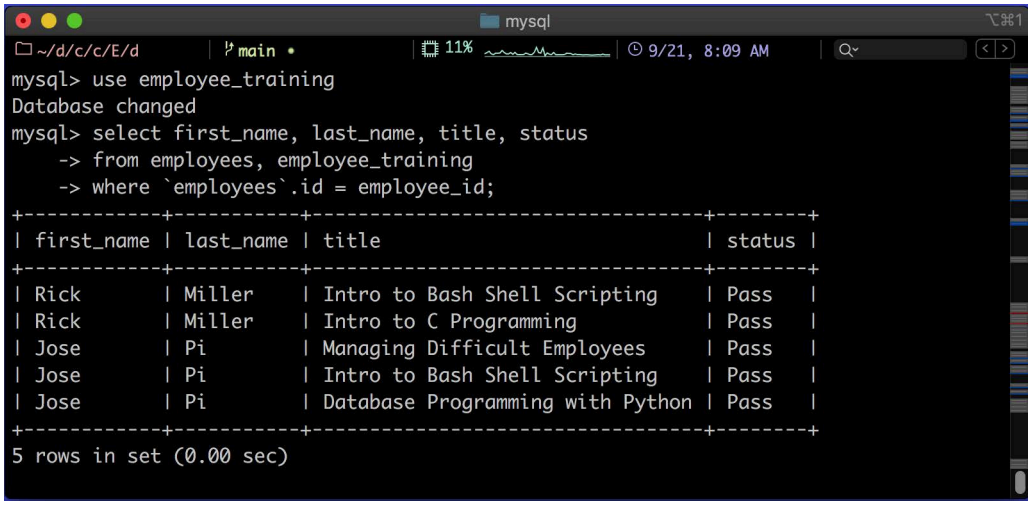

Figure 23-16: Complex Query Involving Two Related Tables

Referring to figure 23-16 — This query performs an `inner join` on two related tables, `employees` and `employee_training`, by comparing the `PRIMARY KEY` (`` `employees`.id``) of the

employees table with the FOREIGN KEY (employee_id) of the employee_training table. The fully-qualified reference to the `employees`.id column is required because both tables have a column named id.

3.1.2 Problems With This Database Structure

The one-to-many relationship between the employees table and the employee_training table will work, but contains quite a bit of repetitive data. If two employees take the same training, the title and description of the training must be entered twice. Unless extraordinary measures are taken to ensure accuracy, eventually mistakes will be made that will introduce data inconsistency.

What's needed is a database structure that separates the notion of a course or unit of training from the notion of attending the training. For example, again referring to figure 23-16 — Rick and Jose both attended training titled Intro to Bash Shell Scripting. In the current database structure, information about that course is repeated for each employee who attends it.

A better approach, and one that significantly reduces data redundancy and enhances data integrity, is to introduce a dedicated table that stores data about courses, and another table that links an employee's attendance of a particular course with dates of attendance and their score. This type of database structure is known as a *many-to-many* or *n-to-n* relationship.

3.2 The Many-To-Many Relationship

Figure 23-17 shows how the new database structure will look.

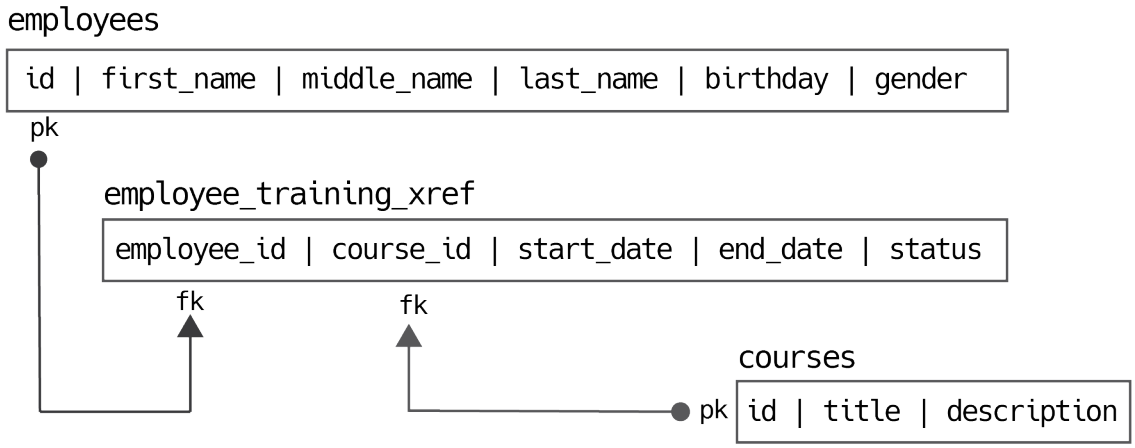

Figure 23-17: Many-to-Many Relationship

Referring to figure 23-17 — The courses table stores data about available courses. Like the employees table, data about a particular course is repeated only once per row. The employee_training_xref table serves to link employees with the courses they have taken. One employee can attend many courses and a course can be attended by many employees.

3.2.1 Modified create_tables.sql Script

Example 23.7 gives the modified create_tables.sql script to create the database structure shown in figure 23-17.

23.7 create_tables.sql

```
/* ************************************************************
   Drop and Create the tables for the employee_training database.
************************************************************** */

-- Switch to employee_training database
USE `employee_training`

-- ----------------------------
-- EMPLOYEES TABLE
-- Drop the table if it exists
DROP TABLE IF EXISTS `employees`;

-- Create the table
CREATE TABLE IF NOT EXISTS `employees` (
    `id` int(11) NOT NULL,
    `first_name` varchar(25) NOT NULL,
    `middle_name` varchar(25) NOT NULL,
    `last_name` varchar(25) NOT NULL,
    `birthday` varchar(25) NOT NULL,
    `gender` char(1) NOT NULL
);

-- Designate the `id` column as the primary key
ALTER TABLE `employees`
  ADD PRIMARY KEY (`id`);

-- Make `id` column auto increment on inserts
ALTER TABLE `employees`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;

-- -------------------------------
-- COURSES TABLE
-- Drop the table if it exists
DROP TABLE IF EXISTS `courses`;

-- Create the table
CREATE TABLE IF NOT EXISTS `courses` (
    `id` int(11) NOT NULL,
    `title` varchar(100) NOT NULL,
    `description` varchar(250) NOT NULL
);

-- Add primary key
ALTER TABLE `courses`
  ADD PRIMARY KEY (`id`);

-- Make `id` column AUTO INCREMENT
ALTER TABLE `courses`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;

-- --------------------------------
-- EMPLOYEE_TRAINING_XREF TABLE
-- Drop employee_training_xref table if it exists
DROP TABLE IF EXISTS `employee_training_xref`;

-- Create employee_training table
CREATE TABLE `employee_training_xref` (
    `employee_id` int(11) NOT NULL,
```

```
59      `course_id` int(11) NOT NULL,
60      `start_date` varchar(25) NOT NULL,
61      `end_date` varchar(25) NOT NULL,
62      `status` varchar(25) NOT NULL
63    );
64
65    -- Create indexes on employee_id and course_id columns
66    ALTER TABLE `employee_training_xref`
67      ADD KEY `employee_training_xref_ibfk_1` (`employee_id`),
68      ADD KEY `employee_training_xref_ibfk_2` (`course_id`);
69
70    -- Add Cascade Delete Constraint on employee_id column
71    ALTER TABLE `employee_training_xref`
72      ADD CONSTRAINT `employee_training_ibfk_1`
73      FOREIGN KEY (`employee_id`) REFERENCES `employees` (`id`)
74      ON DELETE CASCADE
75      ON UPDATE CASCADE;
76
77    -- Add Cascade Delete Constraint on course_id column
78    ALTER TABLE `employee_training_xref`
79      ADD CONSTRAINT `employee_training_ibfk_2`
80      FOREIGN KEY (`course_id`) REFERENCES `courses` (`id`)
81      ON UPDATE CASCADE;
82
```

Referring to example 23.7 — The `courses` table definition section begins on line 32 and is pretty simple since it only consists of three columns. The `employee_training_xref` table definition section begins on line 52. The index definitions for the foreign keys start on line 66, and the foreign key constraints are defined on lines 71 and 78 respectively.

To run this script, you'll first need to run the `create_database.sql`. Launch a terminal, navigate to the project's database folder, and run the following commands:

```
mysql < create_database.sql
mysql < create_tables.sql
```

You can verify the new database structure with phpMyAdmin or via the `mysql` client as shown in figure 23-18.

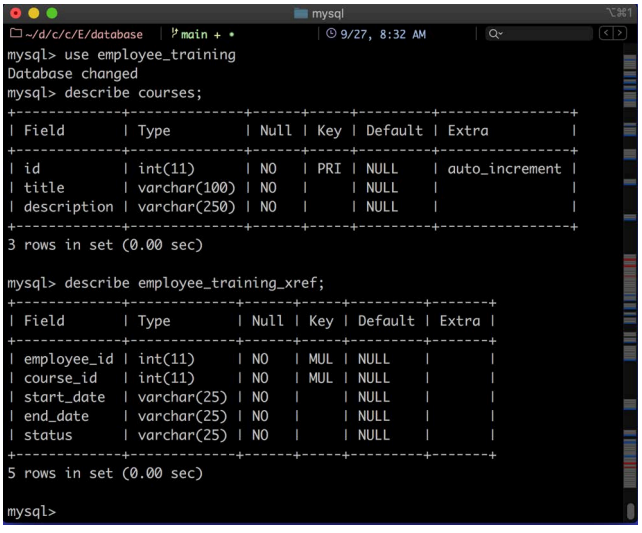

Figure 23-18: Verify New Database Structure

Referring to figure 23-18 — If everything looks good, guess what we need now?

If you spilled your Triple-Shot-Double-Cream-Extra-Caff Latte while shouting "Test Data!", you're right! Example 23.8 gives the modified `insert_test_data.sql` script.

23.8 insert_test_data.sql

```
/* ******************************************************
   Insert test data into the employee_training database.
****************************************************** */

-- Switch to the employee_training database.
USE `employee_training`

-- Insert data into the employees table.
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Rick', 'Warren', 'Miller', '1976-08-20 00:00:00', 'M');
INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
       VALUES ('Jose', 'Miguel', 'Pi', '1985-12-01 00:00:00', 'M');

-- Insert data into the courses table.
INSERT INTO `courses` (title, description)
      values ('Intro to Bash Shell Scripting',
              'How to script the Bash shell.');
INSERT INTO `courses` (title, description)
      values ('Intro to C Programming',
              'Introduces student to C programming language fundamentals.');
INSERT INTO `courses` (title, description)
      values ('Managing Difficult Employees',
              'How to lead through kindness.');
INSERT INTO `courses` (title, description)
      values ('Database Programming with Python',
              'How to develop data-driven apps with Python.');

-- Insert data into the employee_training_xref table.
INSERT INTO `employee_training_xref` (employee_id, course_id, start_date,
                           end_date, status)
      VALUES(1, 1, '2025-09-15 00:00:00', '2025-09-20 00:00:00', 'Pass');
INSERT INTO `employee_training_xref` (employee_id, course_id, start_date,
                           end_date, status)
      VALUES(2, 1, '2025-09-15 00:00:00', '2025-09-20 00:00:00', 'Pass');

```

Referring to example 23.8 — This script first inserts data into the `employees` table. It then populates the `courses` table. Finally, it uses primary key values from those two tables to create rows in the `employee_training_xref` table. Again, I know from experience what the primary key values will be given a clean database and a few rows of test data. However, there is a way to extract the last inserted row id after an `INSERT` statement and use that value to set a variable, which can be used in subsequent SQL statements. Example 23.9 gives the modified `insert_test_data.sql` script.

23.9 insert_test_data.sql

```
1   /* ******************************************************
2      Insert test data into the employee_training database.
3   ******************************************************/
4
5   -- Switch to the employee_training database.
6   USE `employee_training`
7
8   -- Insert data into the employees table.
9   INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
10         VALUES ('Rick', 'Warren', 'Miller', '1976-08-20 00:00:00', 'M');
11  SET @rick_id = LAST_INSERT_ID();
12
13  INSERT INTO `employees` (first_name, middle_name, last_name, birthday, gender)
14         VALUES ('Jose', 'Miguel', 'Pi', '1985-12-01 00:00:00', 'M');
15  SET @jose_id = LAST_INSERT_ID();
16
17  -- Insert data into the courses table.
18  INSERT INTO `courses` (title, description)
19        values ('Intro to Bash Shell Scripting',
20                'How to script the Bash shell.');
21  SET @bash_course_id = LAST_INSERT_ID();
22
23  INSERT INTO `courses` (title, description)
24        values ('Intro to C Programming',
25                'Introduces student to C programming language fundamentals.');
26  INSERT INTO `courses` (title, description)
27        values ('Managing Difficult Employees',
28                'How to lead through kindness.');
29  INSERT INTO `courses` (title, description)
30        values ('Database Programming with Python',
31                'How to develop data-driven apps with Python.');
32
33  -- Insert data into the employee_training_xref table.
34  INSERT INTO `employee_training_xref` (employee_id, course_id, start_date,
35                             end_date, status)
36        VALUES(@rick_id, @bash_course_id, '2025-09-15 00:00:00',
37            '2025-09-20 00:00:00', 'Pass');
38  INSERT INTO `employee_training_xref` (employee_id, course_id, start_date,
39                             end_date, status)
40        VALUES(@jose_id, @bash_course_id, '2025-09-15 00:00:00',
41            '2025-09-20 00:00:00', 'Pass');
42
```

Referring to example 23.9 — After the `INSERT` statement, which begins on line 9, completes, you can retrieve the last `AUTO INCREMENTED` primary key value with a call to the function `LAST_INSERT_ID()`, which is called on line 11 to `SET` a variable named `@rick_id`. I create another variable on line 15 named `@jose_id` and SET its value using the `LAST_INSERT_ID()` function. I do this one more time on line 21 where I define a variable named `@bash_course_id`

and SET its value using the `LAST_INSERT_ID()` function. Finally, I use these variables in the INSERT statements on lines 36 and 40.

To run this script you may as well start from scratch. Launch a terminal and run the following commands:

```
mysql < create_database.sql

mysql < create_tables.sql

mysql < insert_test_data.sql
```

You can verify the data by starting the `mysql` client and entering the following query at the `mysql>` prompt:

```
USE employee_training

SELECT first_name, last_name, title, description
FROM employees, courses, employee_training_xref
WHERE (`employees`.id = employee_id) AND (`courses`.id = course_id);
```

Your results should look similar to that shown in figure 23-19.

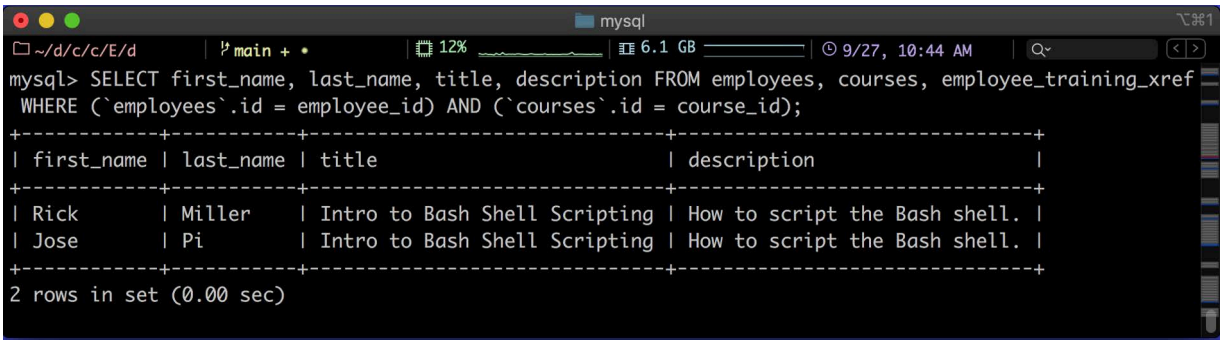

Figure 23-19: Verifying Test Data with Complex SQL Query Across Multiple Tables

Referring to figure 23-19 — If this query works for you, it's time to learn Python database programming.

Quick Review

The power of relational databases lies in their ability to define relationships between tables of which there are two primary types: one-to-many and many-to-many.

In a one-to-many relationship, the primary key of a parent table serves as the foreign key in a child table. This is a useful relationship but can lead to repetitive data and eventual loss of data integrity.

In a many-to-many relationship, a cross-reference, or xref table is introduced, which links two or more tables via primary key and foreign key relationships.

Foreign key constraints can enforce referential integrity. One type of foreign key constraint is a Cascade Delete, where all related child records are deleted upon deletion of the parent record.

4 Creating An Application Framework

In this section, I will show you how to build an application framework that supports essential services including application *configuration*, *settings*, and *logging*. Moving application configuration to separate configuration files increases security and flexibility by avoiding the need to hard-code these values into your Python code. Logging services help you troubleshoot problems by writing debug and error messages to log files for later analysis. Consolidating these services in an application base class makes them uniformly available to application subclasses.

4.1 Application Security

Yes, you need to keep application security foremost in your mind, especially with database programming which necessarily entails IP addresses, host names, database names, user names, and passwords. Some of these items are considered *application secrets*, and they are required to establish a connection to your database. **Do not embed these values in your Python code.** And you never, ever push application secrets to your GitHub repository.

To separate these database connection settings and credentials from the Python code, I'll use the following approach:

- Add database host entry to the local or deployment machine's `etc/hosts` file.
- Store database connection settings in a JSON configuration file.
- Load the designated configuration file into the application on startup.
- Convert the JSON configuration file to a Python dictionary.
- Access configuration settings within the application.

4.2 JSON Configuration File

First, navigate to the `Employee_Training_App` project directory and create a directory named `config`:

```
mkdir config
```

Add the `employee_training_app_config.json` file listed in example 23.10

23.10 employee_training_app_config.json

```
1    {
2       "meta":{
3          "version": "v1",
4          "app_name": "Employee Training",
5          "log_prefix": "employee_training"
6       },
7       "database":{
8          "pool":{
9              "name": "emp_training_app_db_bool",
10             "size": 10,
11             "reset_session": true
12         },
13         "connection":{
14             "config":{
15                "database": "employee_training",
16                "user": "employee_training_user",
17                "host": "localhost",
18                "port": 8889
```

```
            }
        }
    }
}

```

Referring to example 23.10 — The JSON configuration file contains two top-level keys: `"meta"` and `"database"`. The database section defines the `"pool"` and `"connection"` keys. We'll will use both of these keys to configure the database connection pool. But first, we'll need a way to load the configuration file. Let's load it when we run the `main.py` file. It will also be helpful to add a command-line argument that will allow us to designate which configuration file to load on startup. Example 23.11 lists the code for the initial version of the `main.py` file.

23.11 main.py

```
"""Entry point for the Employee Training Application."""

import json
from argparse import ArgumentParser


def main():
    """Entry point."""
    args = configure_and_parse_commandline_arguments()

    if args.configfile:
        config = None
        with open(args.configfile, 'r') as f:
            config = json.loads(f.read())
            print(config)


def configure_and_parse_commandline_arguments():
    """Configure and parse command-line arguments."""
    parser = ArgumentParser(
    prog='main.py',
    description='Start the Employee Training App with a configuration file.',
    epilog='POC: Rick Miller | rick@pulpfreepress.com')

    parser.add_argument('-c','--configfile',
                help="Configuration file to load.",
                required=True)
    args = parser.parse_args()
    return args


if __name__ == "__main__":
    main()

```

Referring to example 23.11 — In this initial version of `main.py`, I'm simply loading the designated JSON configuration file, converting it into a Python dictionary, and printing the resulting dictionary to the console.

To run this program, launch a terminal, navigate to the `Employee_Training_App` project directory and run the following command:

```
pipenv run python src/main.py -c config/employee_training_app_config.json
```

Your output should look similar to that shown in figure 23-20.

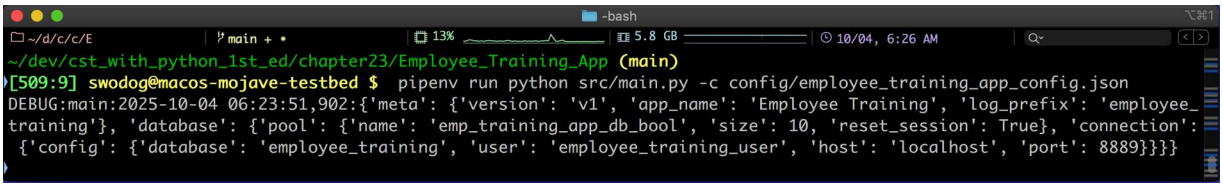

Figure 23-20: Printing the `config` Dictionary to the Console

Referring to figure 23-20 — The `-c` command-line switch supplies the path to the required configuration file.

4.3 Application Settings and Logging

A lot of things can go horribly wrong when you're connecting and interacting with a database. It's helpful to enable logging so you can write error messages to log files for further analysis. It's also helpful to be able to set the log level and the name of the log file. Again, you do not want these setting to be hard coded into the application, not because they're secret, but because you'll want to change them without repackaging and redeploying the application.

4.3.1 Application Settings

I'll start first by creating a `Settings` class, which I'll use to configure and automatically create an `app_settings.json` file. Example 23.12 gives the code for the `settings.py` file.

23.12 settings.py

```
1    """Manage application settings."""
2
3    import json
4    import platform
5    from pathlib import Path
6
7    class Settings():
8        """Manage application settings."""
9
10       def __init__(self, default_settings_filename:str='app_settings.json'):
11           """Initialize instance."""
12           self._default_settings_filename = default_settings_filename
13
14
15       def create_settings_json_file(self, filename:str='app_settings.json')->dict:
16           """Create default settings file if none exists."""
17           if not isinstance(filename, str):
18               filename = self._default_settings_filename
19
20           settings = {}
21
22           match platform.system():
23               case 'Windows':
24                   settings['logs_dir'] = 'logs'
25                   settings['log_filename'] = 'app.log'
26                   settings['log_level'] = 'debug'
27                   settings['log_to_console'] = True
28                   settings['log_to_file'] = True
29                   settings['deployed_to_production'] = False
30
31               case _:
```

```
32                  settings['logs_dir'] = 'logs'
33                  settings['log_filename'] = 'app.log'
34                  settings['log_level'] = 'debug'
35                  settings['log_to_console'] = True
36                  settings['log_to_file'] = True
37                  settings['deployed_to_production'] = False
38          try:
39              with open(filename, 'w') as f:
40                  f.write(json.dumps(settings))
41          except Exception as e:
42              raise Exception(f'{e}')
43          return settings
44
45
46      def read_settings_file_from_location(self,
47                                           filename:str='app_settings.json')->dict:
48          """Read settings file and return dictionary.
49          If settings file does not exist create it.
50          """
51          settings = {}
52          try:
53              with open(filename, 'r') as f:
54                  settings = json.loads(f.read())
55          except Exception as e:
56              settings = self.create_default_settings_json_file(filename)
57
58          return settings
59
```

Referring to example 23.12 — The `Settings` class is used to create and load an application settings file. The default application settings filename is set to `app_settings.json`. You'll see how this class works here in a bit, but essentially, if the designated settings file is not present, one will be automatically created and populated with the values appropriate for either Windows or macOS/Linux. In this example, both sets of property values assume directories are within the application's working directory. (*Remember, a Python application's working directory is the directory from which the application is executed unless set otherwise during execution.*)

Note that the application settings include the name of the logs directory, the log file name, logging level, and whether or not logs should be written to the console and/or to a file. Also, I have a `deployed_to_production` property which can be used within the application as required.

4.3.2 Application Logging

The Python standard library provides a comprehensive logging capability. I like to wrap this functionality in a `LoggingService` convenience class listed in example 23.13.

23.13 logging.py

```
1   """Provides LoggingService convenience class for application logging."""
2
3   import logging
4   import logging.handlers
5   from employee_training.settings import Settings
6   import os
7
8   class LoggingService():
9       """Provides logging services."""
10
11      def __init__(self, class_name:str, logfile_prefix_name:str=None)->None:
```

```
12            """Initialize instance."""
13
14            self._logger = logging.getLogger(class_name)
15            self._logger.propagate = False
16            self._settings_dict = Settings().read_settings_file_from_location()
17            self._logfile_prefix_name = logfile_prefix_name
18            self.log_level = logging.ERROR
19
20            if self._settings_dict['log_level'] == 'notset':
21                self._logger.setLevel(logging.NOTSET)
22                self.log_level = logging.NOTSET
23            elif self._settings_dict['log_level'] == 'debug':
24                self._logger.setLevel(logging.DEBUG)
25                self.log_level = logging.debug
26            elif self._settings_dict['log_level'] == 'info':
27                self._logger.setLevel(logging.INFO)
28                self.log_level = logging.INFO
29            elif self._settings_dict['log_level'] == 'warning':
30                self._logger.setLevel(logging.WARNING)
31                self.log_level = logging.WARNING
32            elif self._settings_dict['log_level'] == 'error':
33                self._logger.setLevel(logging.ERROR)
34                self.log_level = logging.ERROR
35            elif self._settings_dict['log_level'] == 'critical':
36                self._logger.setLevel(logging.CRITICAL)
37                self.log_level = logging.CRITICAL
38            else:
39                self._logger.setLevel(logging.ERROR)
40                self.log_level = logging.ERROR
41
42            self._formatter = \
43                  logging.Formatter('%(levelname)s:%(name)s:%(asctime)s:%(message)s')
44
45            if not self._logger.handlers:
46                if self._settings_dict['log_to_console']:
47                    self._ch = logging.StreamHandler()
48                    self._ch.setLevel(logging.DEBUG)
49                    self._ch.setFormatter(self._formatter)
50                    self._logger.addHandler(self._ch)
51
52                if self._settings_dict['log_to_file']:
53                    log_file = os.path.join(self._settings_dict['logs_dir'],
54                                f"{self._logfile_prefix_name}_" \
55                                f"{self._settings_dict['log_filename']}")
56                    self._fh = logging.handlers.TimedRotatingFileHandler(log_file,
57                                when='midnight', backupCount=20)
58                    self._fh.setLevel(logging.DEBUG)
59                    self._fh.setFormatter(self._formatter)
60                    self._logger.addHandler(self._fh)
61
62
63        def log_debug(self, message):
64            """Log to debug."""
65            self._logger.debug(message)
66
67        def log_error(self, message):
68            """Log to error."""
69            self._logger.error(message)
70
```

```
71      def log_info(self, message):
72          """Log to info."""
73          self._logger.info(message)
74
75      def log_warning(self, message):
76          """Log to warning."""
77          self._logger.warning(message)
78
79      def log_critical(self, message):
80          """Log to critical."""
81          self._logger.critical(message)
82
```

Referring to example 23.13 — Essentially, the `LoggingService` class creates a `logger`, then reads the application settings file to get various logger properties. The `__init__()` method provides two parameters: `class_name` and `logfile_prefix_name`. On line 20, the big, honkin' nested `if` statement sets the logging level based on the level found in the application settings file. Next, line 42 sets the log output format. The `if` statement starting on line 45 configures the log handlers based on the values found in the application settings file. Finally, I provide convenience methods for each logging level. To learn more about Python logging see *https://docs.python.org/3/library/logging.html*.

4.3.3 ApplicationBase Class

I like to deliver common application services in a base class from which other classes can inherit. The code for the `ApplicationBase` class is listed in example 23.14.

23.14 application_base.py

```
1   """Implements behavior common to all application classes.."""
2
3   from abc import ABC, abstractmethod
4   from employee_training.logging import LoggingService
5   from employee_training.settings import Settings
6
7   class ApplicationBase(ABC):
8       """Implements ApplicationBase class."""
9       def __init__(self, subclass_name:str, logfile_prefix_name:str)->None:
10          """Instantiate instance."""
11          self._settings = Settings().read_settings_file_from_location()
12          self._logger = LoggingService(subclass_name, logfile_prefix_name)
13
```

Referring to example 23.14 — The `ApplicationBase` class inherits from `ABC` and provides access to application settings and logging to its subclasses. The `__init__()` method defines two parameters: `subclass_name` and `logfile_prefix_name`. It reads the settings file and then creates the LoggingService. OK, let's now turn our attention to the Persistence Layer and test this application framework.

4.4 Persistence Layer

If you're saying, "Finally, we're getting to the heart of the matter!", I don't blame you. However, before we get into Python database programming we need to test the application framework, and to do that we need to start with baby steps. In the `persistence_layer` directory create a file named `mysql_persistence_wrapper.py` whose listing is shown in example 23.15.

23.15 mysql_persistence_wrapper.py

```
1    """Defines the MySQLPersistenceWrapper class."""
2
3    from employee_training.application_base import ApplicationBase
4    from mysql.connector.pooling import (MySQLConnectionPool)
5
6    class MySQLPersistenceWrapper(ApplicationBase):
7        """Implements the MySQLPersistenceWrapper class."""
8
9        def __init__(self, config:dict)->None:
10           """Initializes object. """
11           self._config_dict = config
12           self.META = config["meta"]
13           self.DATABASE = config["database"]
14           super().__init__(subclass_name=self.__class__.__name__,
15                    logfile_prefix_name=self.META["log_prefix"])
16           self._logger.log_debug(f'It works!')
17
```

Referring to example 23.15 — The `__init__()` method defines one parameter named `config`, which is a dictionary representing configuration settings. In this case, it expects to find two keys `"meta"` and `"database"`, which I assign to `self.META` and `self.DATABASE` for convenience and clarity. On line 14, I call the base class initializer with the current class name and the `"log_prefix"` defined in the `"meta"` section of the configuration file. Finally, on line 16, I'm testing the logging service by writing a debug message, which should write to both the console and to the designated log file in the project's `logs` directory. Now, to test this class, I'll modify the `main.py` file as shown in example 23.16.

23.16 main.py

```
1    """Entry point for the Employee Training Application."""
2
3    import json
4    from argparse import ArgumentParser
5    from employee_training.persistence_layer.mysql_persistence_wrapper \
6        import MySQLPersistenceWrapper
7
8
9    def main():
10       """Entry point."""
11       args = configure_and_parse_commandline_arguments()
12
13       if args.configfile:
14           config = None
15           with open(args.configfile, 'r') as f:
16               config = json.loads(f.read())
17
18           db = MySQLPersistenceWrapper(config)
19
20
21   def configure_and_parse_commandline_arguments():
22       """Configure and parse command-line arguments."""
23       parser = ArgumentParser(
24       prog='main.py',
25       description='Start the Employee Training App with a configuration file.',
26       epilog='POC: Rick Miller | rick@pulpfreepress.com')
27
28       parser.add_argument('-c','--configfile',
29                    help="Configuration file to load.",
```

```
30                      required=True)
31      args = parser.parse_args()
32      return args
33
34
35  if __name__ == "__main__":
36      main()
37
```

Referring to example 23.16 — On line 5, I import the `MySQLPersistenceWrapper` class. On line 18, I create an instance of `MySQLPersistenceWrapper` and pass in the `config` dictionary. If everything works as expected (*The hope-filled moan of a late-night coder sans petite amie!*) you should see a debug log message written to the console and to the designated log file. To run this program, navigate to the `Employee_Training_App` directory and type the following command:

```
pipenv run python src/main.py -c config/employee_training_app_config.json
```

Your console output should look similar to that shown in figure 23-21.

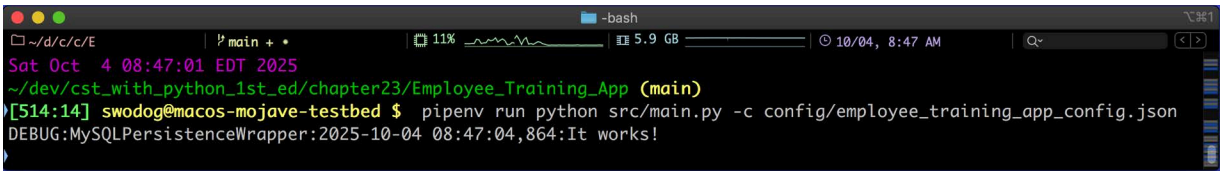

Figure 23-21: Console Logging Works!

Referring to figure 23-21 — It appears console logging works! You should also see a log file in the `logs` directory as shown in figure 23-22.

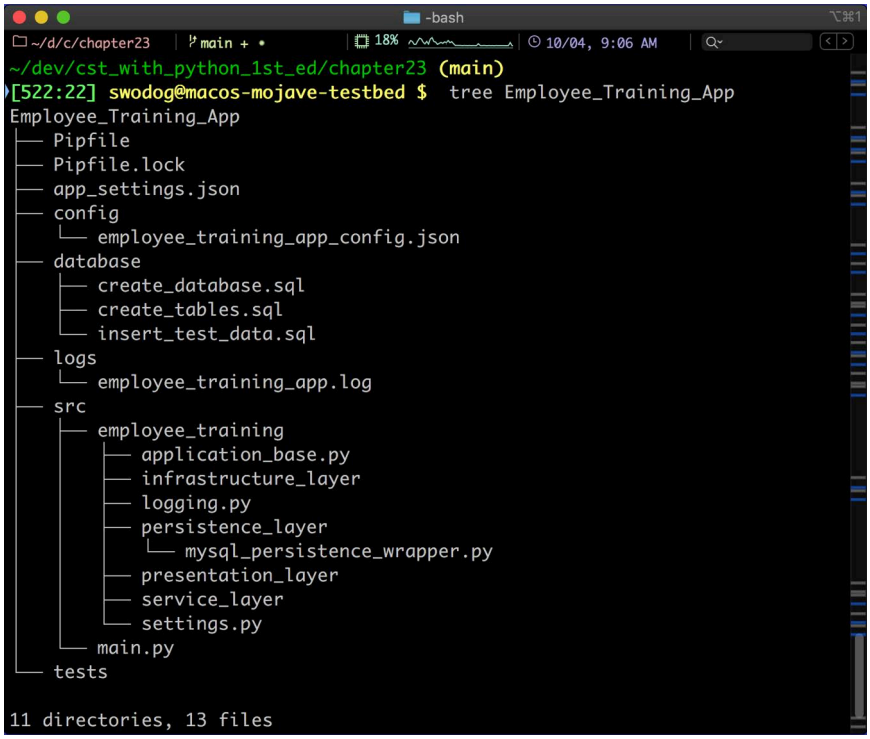

Figure 23-22: Project Directory Structure with Logs

Referring to figure 23-22 — If you `cat` the log file you should see a line similar to the following:

```
cat logs/employee_training_app.log
DEBUG:MySQLPersistenceWrapper:2025-10-04 09:05:05,369:It works!
```

Of course, your date and time values will be different. OK, let's do some Python database programming.

Quick Review

An application framework delivers essential application configuration, settings, and logging services. Moving hardcoded values out of the Python source code and into separate configuration files increases application security and flexibility. Being able to write debug and error messages to log files enhances troubleshooting. Consolidating application framework services into a base class makes them available to subclasses.

5 Python Database Ops

Now that we have a suitable application framework upon which to build, it's time to access the database with Python. To do this, you have to solve a small handful of problems. But once you see your first example working, the rest, as the saying goes, is a piece of cake. Generally speaking, to get your first example working, you must complete the following tasks:

- Create a database connection pool.
- Define an SQL query statement string constant which you will use to create a *prepared statement*.
- Define a method that executes the query and returns the results.
- Call the method and print the results to the console.

If you manage to successfully complete these four steps, you will have solved easily 90% of the hard problems related to Python database programming. However, before proceeding, make sure your MySQL server is running and you have run the database scripts discussed earlier in this chapter including inserting test data. This leads me to a very important Pro Tip:

PRO TIP: Make sure your database server is running.

I've lost count of how many times I have tried to run an application and received a database connection error only to realize my database server wasn't running. You might think this is really a Newbie Tip but trust me, pros have scratched their heads wondering why they can't connect to their database only to discover the server isn't running.

5.1 Create `employee_training_user` Database User

Before proceeding, you'll need to create a *database user* the application can use to connect to the database. If you look at the configuration file shown in example 23.10, the expected database user name is `employee_training_user`. Example 23.17 gives the code for a `create_user.sql` script you can use to create the user.

23.17 create_user.sql

```
/* ****************************************
  Drop and create the employee_training_user
**************************************** */

-- Drop user if exists
DROP USER IF EXISTS 'employee_training_user'@'%';

-- Create user if not exists
CREATE USER IF NOT EXISTS 'employee_training_user'@'%';
GRANT ALL PRIVILEGES ON *.* TO 'employee_training_user'@'%';
ALTER USER 'employee_training_user'@'%'
    REQUIRE NONE WITH MAX_QUERIES_PER_HOUR 0
    MAX_CONNECTIONS_PER_HOUR 0
    MAX_UPDATES_PER_HOUR 0
    MAX_USER_CONNECTIONS 0;
GRANT ALL PRIVILEGES ON `employee\_training\_user\_%`.*
    TO 'employee_training_user'@'%';
GRANT ALL PRIVILEGES ON `employee\_training`.*
    TO 'employee_training_user'@'%' WITH GRANT OPTION;

```

Referring to example 23.17 — This script grants the `employee_training_user` sweeping powers. I usually do this during development and then lock down the user to only the required rights prior to deployment into a production environment.

To run this script, launch a terminal, navigate to the project's `database` directory, and type the following command:

```
mysql < create_user.sql
```

You can verify the user was created successfully by launching the `mysql` client and at the `mysql>` prompt type the following command:

```
SELECT user, host FROM mysql.user;
```

The console output should look similar to that shown in figure 23-23.

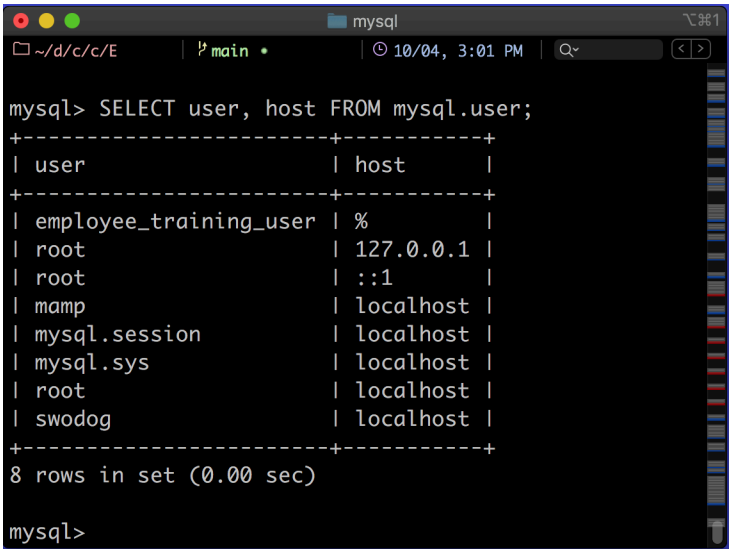

Figure 23-23: Verifying `employee_training_user` Exists

5.2 Modify `MySQLPersistenceWrapper` Class

Alright, edit the `mysql_persistence_wrapper.py` file and make it look like example 23.18.

23.18 mysql_persistence_wrapper.py

```
1    """Defines the MySQLPersistenceWrapper class."""
2
3    from employee_training.application_base import ApplicationBase
4    from mysql import connector
5    from mysql.connector.pooling import (MySQLConnectionPool)
6    import json
7
8    class MySQLPersistenceWrapper(ApplicationBase):
9        """Implements the MySQLPersistenceWrapper class."""
10
11       def __init__(self, config:dict)->None:
12           """Initializes object. """
13           self._config_dict = config
14           self.META = config["meta"]
15           self.DATABASE = config["database"]
16           super().__init__(subclass_name=self.__class__.__name__,
17                    logfile_prefix_name=self.META["log_prefix"])
18
19           # Database Configuration Constants
20           self.DB_CONFIG = {}
21           self.DB_CONFIG['database'] = \
22               self.DATABASE["connection"]["config"]["database"]
23           self.DB_CONFIG['user'] = self.DATABASE["connection"]["config"]["user"]
24           self.DB_CONFIG['host'] = self.DATABASE["connection"]["config"]["host"]
25           self.DB_CONFIG['port'] = self.DATABASE["connection"]["config"]["port"]
26
27           self._logger.log_debug(f'DB Connection Config Dict: {self.DB_CONFIG}')
28
29           # Database Connection
30           self._connection_pool = \
31               self._initialize_database_connection_pool(self.DB_CONFIG)
32
33           # SQL Query Consants
34           self.SELECT_ALL_EMPLOYEES = \
35               f"SELECT id, first_name, middle_name, last_name, birthday, gender " \
36               f"FROM employees"
37
38
39       def select_all_employees(self)->list:
40           """Returns a list of all employee rows."""
41           cursor = None
42           results = None
43           try:
44               connection = self._connection_pool.get_connection()
45               with connection:
46                   cursor = connection.cursor()
47                   with cursor:
48                       cursor.execute(self.SELECT_ALL_EMPLOYEES)
49                       results = cursor.fetchall()
50
51               return results
52
53           except Exception as e:
54               self._logger.log_error(f'Problem with select_all_employees(): {e}')
```

```
55
56
57      ##### Private Utility Methods #####
58
59      def _initialize_database_connection_pool(self, config:dict):
60          """Initializes database connection pool."""
61          try:
62              self._logger.log_debug(f'Creating connection pool...')
63              cnx_pool = \
64                  MySQLConnectionPool(pool_name = self.DATABASE["pool"]["name"],
65                      pool_size=self.DATABASE["pool"]["size"],
66                      pool_reset_session=self.DATABASE["pool"]["reset_session"],
67                      **config)
68              self._logger.log_debug(f'Connection pool successfully created!')
69              return cnx_pool
70          except connector.Error as err:
71              self._logger.log_error(f'Problem creating connection pool: {err}')
72              self._logger.log_error(f'Check DB cnfg:\n{json.dumps(self.DATABASE)}')
73          except Exception as e:
74              self._logger.log_error(f'Problem creating connection pool: {e}')
75              self._logger.log_error(f'Check DB conf:\n{json.dumps(self.DATABASE)}')
76
```

Referring to example 23.18 — This example implements three of the four earlier-defined tasks. Let's start with the `initialize_database_connection_pool()` method which begins on line 59. It takes a `config` dictionary argument, which, along with the database pool configuration settings, is used to create a `MySQLConnectionPool`. If the connection pool is successfully created, the method returns a reference to the connection pool: `cnx_pool`. If you forget to start the server or some other dreadful event prevents the connection pool from being created, an attempt is made to log some helpful hints about what to check.

Moving to lines 4 and 5 at the top of the example, I'm importing both the `mysql connector` module and the `MySQLConnectionPool` class. Then, on lines 20 through 25, I'm using the database connection configuration settings to create the configuration dictionary which I pass to the `initialize_database_connection_pool()` method on line 31.

On line 34, I create an SQL query string constant, `self.SELECT_ALL_EMPLOYEES`, which, as its name implies, will select all employees from the database. Note how I have broken the declaration over multiple lines for clarity. I did this partly so the code fits into the margins of this book, but I also like to break the SQL queries on `SELECT`, `FROM`, `WHERE` boundaries.

Finally, I use this query in the `select_all_employees()` method whose definition begins on line 39. To execute the query, a `connection` must be obtained from the connection pool. Then, you must get a `cursor` from the `connection`, and call the `cursor.execute()` method to execute the `self.SELECT_ALL_EMPLOYEES` query string. You must then call the `cursor.fetchall()` method to get the `results`. And don't forget to return the `results`! The `results` are a `list` of row tuples returned by the query.

OK, moment of truth. Let's test the revised `MySQLPersistenceWrapper` class in the `main.py` file. Example 23.19 gives the code.

23.19 main.py

```
1   """Entry point for the Employee Training Application."""
2
3   import json
4   from argparse import ArgumentParser
5   from employee_training.persistence_layer.mysql_persistence_wrapper \
6       import MySQLPersistenceWrapper
```

```
7
8    def main():
9        """Entry point."""
10       args = configure_and_parse_commandline_arguments()
11
12       if args.configfile:
13          config = None
14          with open(args.configfile, 'r') as f:
15             config = json.loads(f.read())
16
17          db = MySQLPersistenceWrapper(config)
18          employees_list = db.select_all_employees()
19          for employee in employees_list:
20             print(f'{employee}')
21
22
23   def configure_and_parse_commandline_arguments():
24       """Configure and parse command-line arguments."""
25       parser = ArgumentParser(
26       prog='main.py',
27       description='Start the Employee Training App with a configuration file.',
28       epilog='POC: Rick Miller | rick@pulpfreepress.com')
29
30       parser.add_argument('-c','--configfile',
31                    help="Configuration file to load.",
32                    required=True)
33       args = parser.parse_args()
34       return args
35
36
37   if __name__ == "__main__":
38       main()
39
```

Referring to example 23.19 — First, I import the `MySQLPersistenceWrapper` class. On line 17, after reading the configuration file, I create an instance of `MySQLPersistenceWrapper` and pass the `config` dictionary to the initializer. On line 18, I call the `select_all_employees()` method and print each row of the `employees_list` to the console. To run this example, launch a terminal, navigate to the `Employee_Training_App` project directory, and run the following command:

```
pipenv run python src/main.py -c config/employee_training_app_config.json
```

Your results should look similar to that shown in figure 23-24.

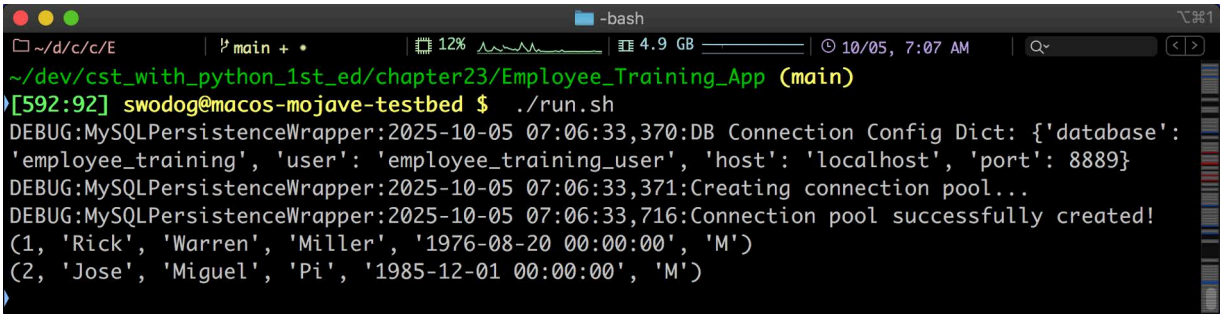

Figure 23-24: Testing `MySQLPersistenceWrapper` in `main.py`

Referring to figure 23-24 — The first three lines output to the console are debug logs that record the database connection configuration settings and the connection pool status. Now you see why I love logging! The last two lines are the results of the query. Notice that each row is a tuple as indicated by the values being surrounded by parentheses vs. square brackets. A tuple is an immutable sequence you can access like a list.

So, everything seems to be working fine. Notice at the prompt, I executed the program with the following command:

```
./run.sh
```

I grew weary, you see, of typing that long `pipenv` command to run the program and created the `run.sh` script listed in example 23.20:

23.20 run.sh

```
1    #!/bin/bash
2
3    pipenv run python src/main.py -c config/employee_training_app_config.json
4
```

Save the `run.sh` script in the project directory and make it executable with the following command:

```
chmod +x run.sh
```

5.2.1 Troubleshooting

If you are running into issues, and you're *sure* your database server is running.You can check the following items:

- Configuration file database connection settings are correct.
- The database `port` is correct: (`8889` for macOS | `3306` Linux and Windows)
- The database `user` is correct and has access to the `employee_training` database.

Examine the logs and look up any database connection error codes you find there.

5.3 Query Parameters

I'd like to add two more methods to the `MySQLPersistenceWrapper` class before *wrapping* up the chapter. (*Oh boy...as the Doctor tells Captain Jack in the movie Master and Commander: "He who would pun would pick a pocket!"*) One method demonstrates how to do a complex query across multiple tables with ambiguous column names and the other demonstrates how to set an SQL query parameter for a prepared statement. The first method will get all employees along with their completed training. The second method will fetch all completed training for a given employee id. Example 23.21 lists the code for the modified `MySQLPersistenceWrapper` class.

23.21 mysql_persistence_wrapper.py

```
1    """Defines the MySQLPersistenceWrapper class."""
2
3    from employee_training.application_base import ApplicationBase
4    from mysql import connector
5    from mysql.connector.pooling import (MySQLConnectionPool)
6    import json
7    import inspect
8
9
10   class MySQLPersistenceWrapper(ApplicationBase):
```

```
11      """Implements the MySQLPersistenceWrapper class."""
12
13      def __init__(self, config:dict)->None:
14          """Initializes object. """
15          self._config_dict = config
16          self.META = config["meta"]
17          self.DATABASE = config["database"]
18          super().__init__(subclass_name=self.__class__.__name__,
19                   logfile_prefix_name=self.META["log_prefix"])
20
21          # Database Configuration Constants
22          self.DB_CONFIG = {}
23          self.DB_CONFIG['database'] = \
24              self.DATABASE["connection"]["config"]["database"]
25          self.DB_CONFIG['user'] = self.DATABASE["connection"]["config"]["user"]
26          self.DB_CONFIG['host'] = self.DATABASE["connection"]["config"]["host"]
27          self.DB_CONFIG['port'] = self.DATABASE["connection"]["config"]["port"]
28
29          self._logger.log_debug(f'DB Connection Config Dict: {self.DB_CONFIG}')
30
31          # Database Connection
32          self._connection_pool = \
33              self._initialize_database_connection_pool(self.DB_CONFIG)
34
35          # SQL Query Consants
36          self.SELECT_ALL_EMPLOYEES = \
37              f"SELECT id, first_name, middle_name, last_name, birthday, gender " \
38              f"FROM employees"
39
40          self.SELECT_ALL_EMPLOYEES_WITH_TRAINING = \
41              f"SELECT `employees`.id, first_name, last_name, title, description, " \
42                    f"start_date, end_date, status " \
43              f"FROM employees, courses, employee_training_xref " \
44              f"WHERE (`employees`.id = employee_id) AND (`courses`.id = course_id)"
45
46          self.SELECT_TRAINING_FOR_EMPLOYEE_ID = \
47              f"SELECT title, description, start_date, end_date, status " \
48              f"FROM courses, employee_training_xref " \
49              f"WHERE (employee_id = %s) AND (`courses`.id = course_id)"
50
51
52      def select_all_employees(self)->list:
53          """Returns a list of all employee rows."""
54          cursor = None
55          results = None
56          try:
57              connection = self._connection_pool.get_connection()
58              with connection:
59                  cursor = connection.cursor()
60                  with cursor:
61                      cursor.execute(self.SELECT_ALL_EMPLOYEES)
62                      results = cursor.fetchall()
63
64              return results
65
66          except Exception as e:
67              self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
68
69
```

```
70      def select_all_employees_with_training(self)->list:
71          """Returns a list of all employee rows with training."""
72          cursor = None
73          results = None
74          try:
75              connection = self._connection_pool.get_connection()
76              with connection:
77                  cursor = connection.cursor()
78                  with cursor:
79                      cursor.execute(self.SELECT_ALL_EMPLOYEES_WITH_TRAINING)
80                      results = cursor.fetchall()
81
82              return results
83
84          except Exception as e:
85              self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
86
87
88      def select_all_training_for_employee_id(self, employee_id:int)->list:
89          """Returns a list of training rows for employee id."""
90          cursor = None
91          results = None
92          try:
93              connection = self._connection_pool.get_connection()
94              with connection:
95                  cursor = connection.cursor()
96                  with cursor:
97                      cursor.execute(self.SELECT_TRAINING_FOR_EMPLOYEE_ID,
98                                ([employee_id]))
99                      results = cursor.fetchall()
100
101             return results
102
103         except Exception as e:
104             self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
105
106
107     ##### Private Utility Methods #####
108
109     def _initialize_database_connection_pool(self, config:dict):
110         """Initializes database connection pool."""
111         try:
112             self._logger.log_debug(f'Creating connection pool...')
113             cnx_pool = \
114                 MySQLConnectionPool(pool_name = self.DATABASE["pool"]["name"],
115                     pool_size=self.DATABASE["pool"]["size"],
116                     pool_reset_session=self.DATABASE["pool"]["reset_session"],
117                     **config)
118             self._logger.log_debug(f'Connection pool successfully created!')
119             return cnx_pool
120         except connector.Error as err:
121             self._logger.log_error(f'Problem creating connection pool: {err}')
122             self._logger.log_error(f'Check DB cnfg:\n{json.dumps(self.DATABASE)}')
123         except Exception as e:
124             self._logger.log_error(f'Problem creating connection pool: {e}')
125             self._logger.log_error(f'Check DB conf:\n{json.dumps(self.DATABASE)}')
126
```

Referring to example 23.21 — Starting on line 40, I've added two new query constants, `self.SELECT_ALL_EMPLOYEES_WITH_TRAINING`, and on line 46, `self.SELECT_TRAINING_FOR_EMPLOYEE_ID`. Two important things to note with these queries. First, if you query across multiple tables, you must fully qualify ambiguous column names. For example, `` `employees`.id `` fully qualifies the `id` column in the `employees` table. Next, the `'%s'` represents a query parameter that must be supplied when the query is executed.

The `select_all_training_for_employee_id()` method that begins on line 88 demonstrates how to set a query parameter. The function defines a corresponding parameter named `employee_id`, which is passed to the `cursor.execute()` method, which spans lines 97 and 98. Note that the query parameter is packaged as a `list` inside of a `tuple`.

One other change I've made to the code is to standardize the way I refer to the executing method for logging purposes by using introspection. See lines 7, 67, 85, and 104.

OK, let's test these new methods in the main module as shown in example 23.22.

23.22 main.py

```
1    """Entry point for the Employee Training Application."""
2
3    import json
4    from argparse import ArgumentParser
5    from employee_training.persistence_layer.mysql_persistence_wrapper \
6        import MySQLPersistenceWrapper
7
8
9    def main():
10       """Entry point."""
11       args = configure_and_parse_commandline_arguments()
12
13       if args.configfile:
14          config = None
15          with open(args.configfile, 'r') as f:
16             config = json.loads(f.read())
17
18          db = MySQLPersistenceWrapper(config)
19          employees_list = db.select_all_employees()
20          for employee in employees_list:
21             print(f'{employee}')
22
23          print('*' * 40)
24          employees_list = db.select_all_employees_with_training()
25          for employee in employees_list:
26             print(f'{employee}')
27
28          print('*' * 40)
29          employees_list = db.select_all_employees()
30          for employee in employees_list:
31             print(f'{employee[1]} {employee[3]}')
32             training_list = db.select_all_training_for_employee_id(employee[0])
33             for training in training_list:
34                training_string = f'\t{training[0]} {training[1]} {training[2]} ' \
35                   f'{training[3]} {training[4]}'
36                print(training_string)
37
38
39   def configure_and_parse_commandline_arguments():
40       """Configure and parse command-line arguments."""
41       parser = ArgumentParser(
```

```
42      prog='main.py',
43      description='Start the Employee Training App with a configuration file.',
44      epilog='POC: Rick Miller | rick@pulpfreepress.com')
45
46      parser.add_argument('-c','--configfile',
47                  help="Configuration file to load.",
48                  required=True)
49      args = parser.parse_args()
50      return args
51
52
53  if __name__ == "__main__":
54      main()
55
```

Referring to example 23.22 — On line 24, I make a call to the `db.select_all_employees_with_training()` method and assign the results to the `employees_list`. I then iterate over the `employees_list` as before and print each row to the console. On line 29, I make a call to `db.get_all_employees()` and then for each `employee` in the `employees_list`, I make a call to `db.get_training_for_employee_id()` method and assign the results to the `training_list`. I then iterate over each row of the `training_list` and print the row by accessing each element in the row tuple. Figure 23-25 shows the results of running this program.

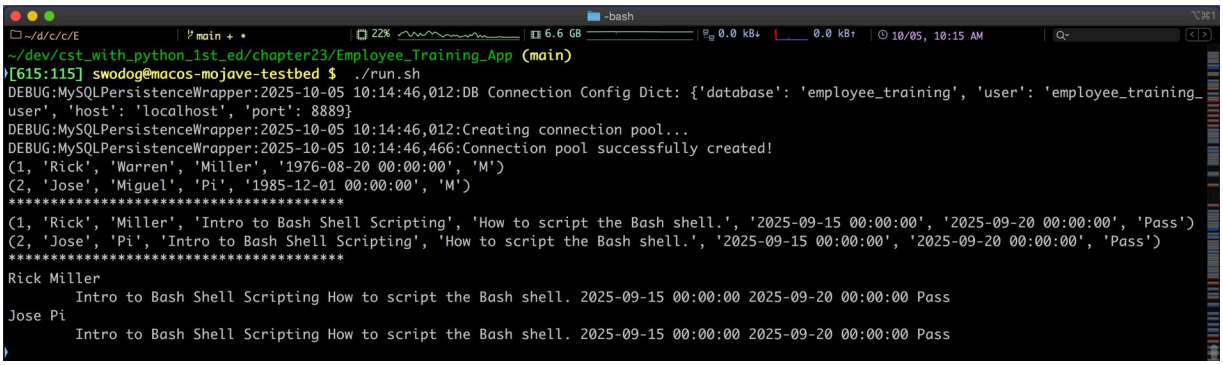

Figure 23-25: Testing Training Queries

Referring to figure 23-25 — That's it for this chapter! If you have everything running as expected, great job. If you're still struggling to get things going, keep at it. Once you solve all the little problems and see it run for the first time, everything you've learned in this chapter will snap into focus.

In *Chapter 24: Scripting The Database*, I'll dive deeper into database scripting and project automation in general. I'll also implement a complete swath through the Employee Training Application Architecture to include the Presentation and Service Layers.

Quick Review

Connecting to a database with Python involves solving a small handful of problems. You must have a properly configured database connection, a database user with access to the target database, a well-formed SQL query string constant for use in a prepared statement, and a method that executes the SQL query and returns the results.

If you have problems connecting to the database, ensure your configuration settings are correct and make sure your database server is running and reachable from the client machine.

When writing complex SQL queries that join multiple tables, fully qualify ambiguous column names by prefixing the table name to the column name.

MySQL query string constants use `'%s'` as parameter placeholders. The value for these placeholders are passed to the `cursor.execute()` method along with the SQL query string.

Favor the use of connection pools to increase code efficiency and reduce the time it takes to acquire a connection to the database.

Use the `with` context manager to automatically release connection and cursor resources.

6 Windows Segmentation Fault

While testing the Employee Training Application on Windows 10 and 11, I encountered a `Segmentation fault` as shown in figure 23-26.

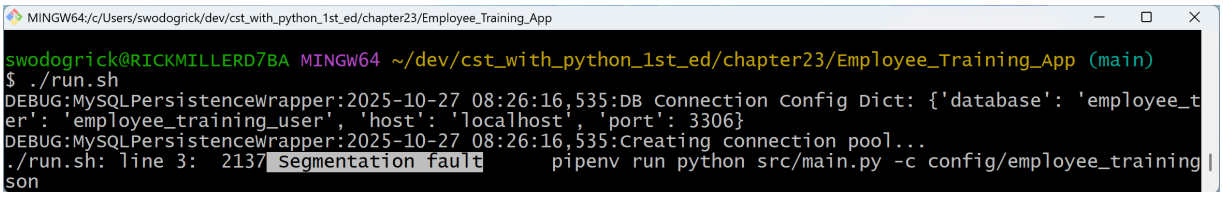

Figure 23-26: Segmentation Fault on Windows 10 & 11

Referring to figure 23-26 — This error occurs in the `MySQLPersistenceWrapper` when attempting to create an instance of `MySQLConnectionPool`. The error is caused by incompatible C language binaries. Note that this error does not affect macOS or Linux users.

6.1 The Fix

The `mysql-connector-python` package is written in pure python. If you are getting this error, you need to set the `use_pure` parameter to `True` (`use_pure=True`) when you call the `MySQLConnectionPool()` initializer. I have modified the `employee_training_app_config.json` file to add the `"use_pure"` key as shown in example 23.23.

23.23 employee_training_app_config.json (modified)

```
1    {
2        "meta":{
3           "version": "v1",
4           "app_name": "Employee Training",
5           "log_prefix": "employee_training"
6        },
7        "database":{
8           "pool":{
9              "name": "emp_training_app_db_bool",
10             "size": 10,
11             "reset_session": true,
12             "use_pure": false
13          },
14          "connection":{
15             "config":{
16                "database": "employee_training",
17                "user": "employee_training_user",
18                "host": "localhost",
19                "port": 3306
```

```
20              }
21          }
22      }
23  }
24
```

Referring to example 23.23 — I have added the `"use_pure"` key on line 12 set its value to `false`. If you receive the `Segmentation fault` error, set it to `true` and try again.

Example 23.24 shows the code for the modified `MySQLPersistenceWrapper` class that uses the `"use_pure"` configuration setting in the `_initialize_database_connection_pool()` method.

23.24 mysql_persistence_wrapper.py (modified)

```
1   """Defines the MySQLPersistenceWrapper class."""
2
3   from employee_training.application_base import ApplicationBase
4   from mysql import connector
5   from mysql.connector.pooling import (MySQLConnectionPool)
6   import json
7   import inspect
8
9
10  class MySQLPersistenceWrapper(ApplicationBase):
11      """Implements the MySQLPersistenceWrapper class."""
12
13      def __init__(self, config:dict)->None:
14          """Initializes object. """
15          self._config_dict = config
16          #self.META = dict(config["meta"])
17          self.META = config["meta"]
18          #self.DATABASE = dict(config["database"])
19          self.DATABASE = config["database"]
20          super().__init__(subclass_name=self.__class__.__name__,
21                   logfile_prefix_name=self.META["log_prefix"])
22
23          # Database Configuration Constants
24          self.DB_CONFIG = {}
25          self.DB_CONFIG['database'] = \
26              self.DATABASE["connection"]["config"]["database"]
27          self.DB_CONFIG['user'] = self.DATABASE["connection"]["config"]["user"]
28          self.DB_CONFIG['host'] = self.DATABASE["connection"]["config"]["host"]
29          self.DB_CONFIG['port'] = self.DATABASE["connection"]["config"]["port"]
30
31          self._logger.log_debug(f'DB Connection Config Dict: {self.DB_CONFIG}')
32
33          # Database Connection
34          self._connection_pool = \
35              self._initialize_database_connection_pool(self.DB_CONFIG)
36
37          # SQL Query Consants
38          self.SELECT_ALL_EMPLOYEES = \
39              f"SELECT id, first_name, middle_name, last_name, birthday, gender " \
40              f"FROM employees"
41
42          self.SELECT_ALL_EMPLOYEES_WITH_TRAINING = \
43              f"SELECT `employees`.id, first_name, last_name, title, description, " \
44                    f"start_date, end_date, status " \
45              f"FROM employees, courses, employee_training_xref " \
46              f"WHERE (`employees`.id = employee_id) AND (`courses`.id = course_id)"
```

```
47
48          self.SELECT_TRAINING_FOR_EMPLOYEE_ID = \
49              f"SELECT title, description, start_date, end_date, status " \
50              f"FROM courses, employee_training_xref " \
51              f"WHERE (employee_id = %s) AND (`courses`.id = course_id)"
52
53
54      def select_all_employees(self)->list:
55          """Returns a list of all employee rows."""
56          cursor = None
57          results = None
58          try:
59              connection = self._connection_pool.get_connection()
60              with connection:
61                  cursor = connection.cursor()
62                  with cursor:
63                      cursor.execute(self.SELECT_ALL_EMPLOYEES)
64                      results = cursor.fetchall()
65
66              return results
67
68          except Exception as e:
69              self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
70
71
72      def select_all_employees_with_training(self)->list:
73          """Returns a list of all employee rows with training."""
74          cursor = None
75          results = None
76          try:
77              connection = self._connection_pool.get_connection()
78              with connection:
79                  cursor = connection.cursor()
80                  with cursor:
81                      cursor.execute(self.SELECT_ALL_EMPLOYEES_WITH_TRAINING)
82                      results = cursor.fetchall()
83
84              return results
85
86          except Exception as e:
87              self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
88
89
90      def select_all_training_for_employee_id(self, employee_id:int)->list:
91          """Returns a list of all training rows for employee id."""
92          cursor = None
93          results = None
94          try:
95              connection = self._connection_pool.get_connection()
96              with connection:
97                  cursor = connection.cursor()
98                  with cursor:
99                      cursor.execute(self.SELECT_TRAINING_FOR_EMPLOYEE_ID,
100                                     ([employee_id]))
101                     results = cursor.fetchall()
102
103             return results
104
105         except Exception as e:
```

```
106             self._logger.log_error(f'{inspect.currentframe().f_code.co_name}: {e}')
107
108
109     ##### Private Utility Methods #####
110
111     def _initialize_database_connection_pool(self, config:dict):
112         """Initializes database connection pool."""
113         try:
114             self._logger.log_debug(f'Creating connection pool...')
115             cnx_pool = \
116                 MySQLConnectionPool(pool_name = self.DATABASE["pool"]["name"],
117                     pool_size=self.DATABASE["pool"]["size"],
118                     pool_reset_session=self.DATABASE["pool"]["reset_session"],
119                     use_pure=self.DATABASE["pool"]["use_pure"],
120                     **config)
121             self._logger.log_debug(f'Connection pool successfully created!')
122             return cnx_pool
123         except connector.Error as err:
124             self._logger.log_error(f'Problem creating connection pool: {err}')
125             self._logger.log_error(f'Check DB cnfg:\n{json.dumps(self.DATABASE)}')
126         except Exception as e:
127             self._logger.log_error(f'Problem creating connection pool: {e}')
128             self._logger.log_error(f'Check DB conf:\n{json.dumps(self.DATABASE)}')
129
```

Referring to example 23.24 — I'm now setting the `"use_pure"` parameter on line 119 to the value of the configuration setting. If you're running the application on Windows, set the `"use_pure"` configuration setting to `true` and run the application. This time everything should work as expected as shown in figure 23-27.

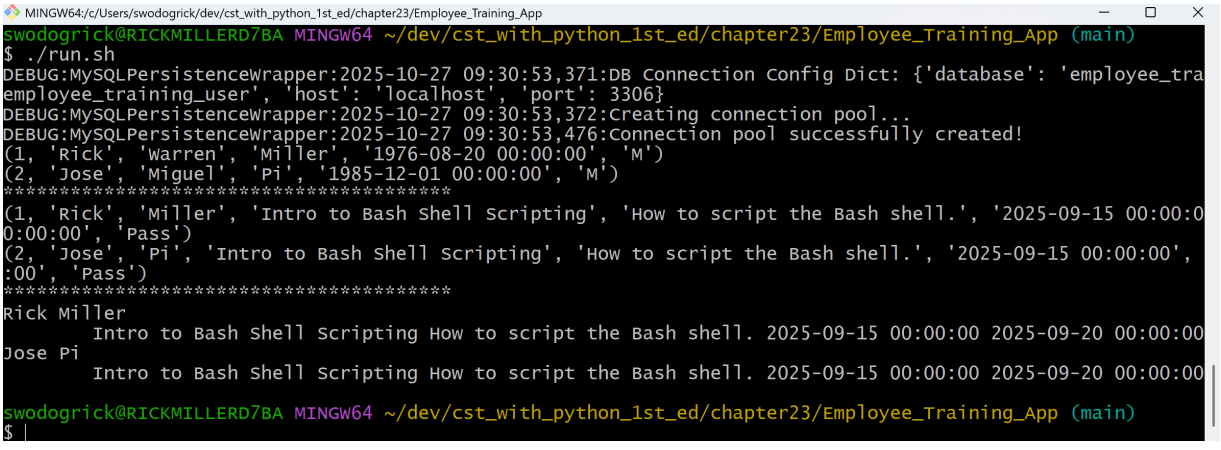

Figure 23-27: Successfully Running Application on Windows 11 with `use_pure` Set to `true`

Referring to figure 23-27 — Ahhh, that's much better. This is a good example of the challenges you will face as a software engineer. Only cross-platform testing will expose these types of issues.

Quick Review

Windows users may receive a Segmentation Fault error when running the Employee Training Application. This error occurs in the `MySQLPersistenceWrapper` when attempting to create an instance of `MySQLConnectionPool`. The error is caused by incompatible C language binaries.

To fix the error, add a `"use_pure"` key to the database pool configuration file and set its value to `true`, then set the `"use_pure"` parameter to the value of the configuration setting when you create an instance of `MySQLConnectionPool` in the `MySQLPersistenceWrapper._initialize_database_connection_pool()` method.

Summary

A multilayered application architecture assigns separate and distinct responsibilities to each application layer. The Employee Training Application's architecture includes the Persistence Layer, Service Layer, Presentation Layer, and the Infrastructure Layer.

Application layers provide a convenient and logical way to organize your project. This goes a long way towards managing physical and conceptual complexity.

To connect to a MySQL database with Python use the `mysql-connector-python` package available from PyPI.

A database management system (DBMS) is a software application that stores data in some form or another, usually on the file system, and provides a suite of software components that enables users to create, manipulate, and delete database objects and data. The term database refers to a related collection of data.

A relational database stores data in relations referred to as tables. A relational database management system (RDBMS) is a software application that enables users to create and manipulate relational databases.

A table is composed of rows and columns. Each column has a name and an associated database type. In most cases it is desirable to be able to uniquely identify each row of data contained within a table. To do this, one or more of the table columns must be designated as the primary key for that table. The important characteristic of a primary key is that its value must be unique for each row.

The power of relational databases derives from their ability to define associations between different tables. One table can be related to another table by the implementation of a foreign key. The primary key of a parent table serves as the foreign key in the related, or child, table. Primary keys and foreign keys can be used together to enforce referential integrity.

Structured Query Language is used to create, manipulate, and delete relational database objects and the data they contain. Although SQL is a standardized database language, each RDBMS vendor is free to add extensions to the language, which essentially renders the language non-portable between different database products.

SQL comprises four sub-languages, which group commands according to functionality: Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), which is used to grant or revoke user rights and privileges on database objects, and Transaction Control Language (TCL), which is used to manage transactions. A transaction is a series of

database commands that must all execute to completion before the operation can be considered complete, or, in case of failure, the commands must all be rolled back.

Data Definition Language (DDL) includes the `CREATE`, `USE`, `ALTER`, and `DROP` commands. Data Manipulation Language (DML) includes the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` commands.

The power of relational databases lies in their ability to define relationships between tables of which there are two primary types: one-to-many and many-to-many.

In a one-to-many relationship, the primary key of a parent table serves as the foreign key in a child table. This is a useful relationship but can lead to repetitive data and eventual loss of data integrity.

In a many-to-many relationship, a cross-reference, or xref table is introduced, which links two or more tables via primary key and foreign key relationships.

Foreign key constraints can enforce referential integrity. One type of foreign key constraint is a Cascade Delete, where all related child records are deleted upon deletion of the parent record.

An application framework delivers essential application configuration, settings, and logging services. Moving hardcoded values out of the Python source code and into separate configuration files increases application security and flexibility. Being able to write debug and error messages to log files enhances troubleshooting. Consolidating application framework services into a base class makes them available to subclasses.

Connecting to a database with Python involves solving a small handful of problems. You must have a properly configured database connection, a database user with access to the target database, a well-formed SQL query string constant for use in a prepared statement, and a method that executes the SQL query and returns the results.

If you have problems connecting to the database, ensure your configuration settings are correct and make sure your database server is running and reachable from the client machine.

When writing complex SQL queries that join multiple tables, fully qualify ambiguous column names by prefixing the table name to the column name.

MySQL query string constants use `'%s'` as parameter placeholders. The value for these placeholders are passed to the `cursor.execute()` method along with the SQL query string.

Favor the use of connection pools to increase code efficiency and reduce the time it takes to acquire a connection to the database.

Use the `with` context manager to automatically release connection and cursor resources.

Windows users may receive a Segmentation Fault error when running the Employee Training Application. This error occurs in the `MySQLPersistenceWrapper` when attempting to create an instance of `MySQLConnectionPool`. The error is caused by incompatible C language binaries.

To fix the error, add a `"use_pure"` key to the database pool configuration file and set its value to `true`, then set the `"use_pure"` parameter to the value of the configuration setting when you create an instance of `MySQLConnectionPool` in the `MySQLPersistenceWrapper._initialize_database_connection_pool()` method.

Skill-Building Exercises

1. **Relational Database Vocabulary:** In your own words, state the purpose and use of each of the terms: Relational Database Management System (RDBMS), Relational Database, Table, Primary Key, Foreign Key, Foreign Key Constraint.

2. **Work Chapter Examples:** Work through the chapter examples, beginning with project structure setup, database scripts, and execute queries from the `mysql` client.

3. **Additional Test Data:** Modify the `insert_test_data.sql` script listed in example 23.8. Add more employees, more courses, and more `employee_training_xref` table entries. Recreate the `employee_training` database by executing the database scripts. Use the `mysql` client to query the database.

4. **Practice Writing SQL Queries:** Run the `insert_test_data.sql` script you modified in the previous exercise. Practice writing SQL queries using the `mysql` client. Start by using SELECT, FROM, WHERE to query data. Use the INSERT statement to add data. Use the UPDATE statement to modify existing data, and use the DELETE statement to delete data. (*Don't be afraid to delete data because you can always recreate the database by running the database scripts.*)

5. **Explore SQL Functions:** Practice using the SQL aggregate functions `COUNT`, `AVG`, and `GROUP BY`.

6. **MySQL Documentation Research:** Research the MySQL documentation listed in the References section. Study the various MySQL data types, functions, and SQL sub-languages DDL, DML, DCL, and DTL.

7. **Further Study:** Obtain the PDF copy of the book *Database Systems: A Practical Approach to Design, Implementation, and Management, Sixth Edition* and read chapters 1 through 19.

8. **Research Normal Forms:** Although not formally discussed in this chapter, the final many-to-many `employee_training` database structure is in 3rd Normal Form. Research how to evolve a database design starting from 1st normal form to 3rd normal form. What are the advantages of normalizing a database? How about disadvantages? When might you want to denormalize a database?

9. **Relational Table Joins:** Research the various ways to join related tables. Note the difference between the `INNER JOIN`, `LEFT JOIN`, and `RIGHT JOIN` and when you would choose one type of join over another. Practice using different types of joins on the `employee_training` database.

10. **The `mysql-connector-python` Package:** Research the *mysql-connector-python* package and the *PEP 249 — Python Database API Specification 2.0*.

Suggested Projects

1. **Asynchronous Connectivity:** The `mysql-connector-python` package support asynchronous connectivity via the `mysql.connector.aio` package. Study the documentation located here: *https://dev.mysql.com/doc/connector-python/en/connector-python-asyncio.html*, and modify the `MySQLPersistenceWrapper` class to use asynchronous database operations.

2. **University Records Management System:** Using the techniques described in this chapter, design and implement a University Records Management System. Ensure your design supports generating student transcripts.

3. **Auto Parts Store Inventory Management System:** Using the techniques described in this chapter, design and implement an Auto Parts Store Inventory Management System.

4. **Embedded Relational Database To-Do List:** Research SQLite. Design and implement a To-Do list application that allows user to create, update, and delete a task list.

5. **PostgreSQL:** Swap out MySQL for PostgreSQL. Create another set of database scripts and a new persistence layer class named `PostgreSQLPersistenceWrapper`.

6. **MariaDB:** Swap out MySQL for MariaDB. MariaDB is supposed to be a MySQL compatible database. What changes, if any, must be made to the database scripts and MySQLPersistence-Wrapper class to work with MariaDB.

7. **Employee Time Tracker:** Modify the `employee_training` application discussed in this chapter to include an employee time tracking feature. Create a table to store project data. Create a cross reference table that supports a many-to-many relationship between employees and the projects they have worked on. The cross reference table should include the project number, date, and the hours worked on the project.

8. **Employee Pictures:** Modify the `employee_training` database to store and retrieve employee pictures.

9. **Object-Relational Mapping (ORM) Frameworks:** Research SQLModel: *https://sqlmodel.tiangolo.com*. Modify the `employee_training` persistence layer to use SQLModel and Pydantic *https://docs.pydantic.dev/latest/* data validation.

10. **Assessment of Object-Relational Mapping (ORM) Frameworks:** After completing Suggested Project 9 above, which do you prefer: hand-coding the persistence layer, or letting the ORM framework do the heavy-lifting for you? Do you foresee any disadvantages to using the ORM framework? How do you think the ORM framework would perform with complex database schemas?

Self-Test Questions

1. (True/False) Structured Query Language (SQL) is 100% portable across different vendor database management systems.

2. What is the primary benefit of adopting a multilayered application architecture?

3. What's the purpose of a primary key?

4. What's the purpose of a foreign key?

5. In a complex query involving multiple tables with ambiguous table names, how do you resolve the ambiguity?

6. What's the purpose of `'%s'` in an SQL string constant?

7. State several benefits of using database connection pools vs. non-pooled connections?

8. What would be the effects of a Cascade Delete foreign key constraint on the related table rows when deleting a row from the parent table?

9. What function would you use to retrieve the last inserted primary key value in an SQL script?

10. Why might you want to structure a set of tables to use a many-to-many relationship vs. a one-to-many?

11. List and describe the four SQL sub-languages.

12. What SQL statement would you use to add a row of data to a table?

13. Which command is used to switch to a particular database before interacting with it via the MySQL client or within an SQL script.

14. What does the UPDATE command return if it successfully executes?

15. If you have trouble connecting to a database with Python, what's the very first thing you should check?

References

MySQL Documentation Site, *https://dev.mysql.com/doc/*

The mysql-connector-python, *https://pypi.org/project/mysql-connector-python/*

The MySQL System Schema, *https://dev.mysql.com/doc/refman/8.4/en/system-schema.html*

MySQL Data Types, *https://dev.mysql.com/doc/refman/8.4/en/data-types.html*

MySQL Functions and Operators, *https://dev.mysql.com/doc/refman/8.4/en/functions.html*

MySQL Data Manipulation Statements, *https://dev.mysql.com/doc/refman/8.4/en/sql-data-manipulation-statements.html*

MySQL Data Definition Statements, *https://dev.mysql.com/doc/refman/8.4/en/sql-data-definition-statements.html*

Python Logging Module, *https://docs.python.org/3/library/logging.html*

PEP 249 — Python Database API Specification v2.0

MariaDB, *https://mariadb.org*

SQLModel, *https://sqlmodel.tiangolo.com*

Pydantic, *https://docs.pydantic.dev/latest/*

Thomas Connolly & Carolyn Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management, Sixth Edition*, Pearson, ISBN 13: 978-1-292-061184

Notes