Chapter 16. Databases

Modern software applications, particularly on the web, use state in order to function. *State* is a way to represent the current condition of the application for a specific request—who is logged in, what page they're on, any preferences they've configured.

Typically, code is written to be more or less stateless. It will function the same way regardless of the state of the user's session (which is what makes system behavior predictable within an application for multiple users). When a web application is deployed, it's done so again in a stateless manner.

But state is vital for keeping track of user activity and evolving the way the application behaves for the user as they continue to interact with it. In order for an otherwise stateless piece of code to be aware of state, it must retrieve that state from somewhere.

Typically, this is done through the use of a database. Databases are efficient ways to store structured data. There are generally four kinds of databases you will work with in PHP: relational databases, key-value stores, graph databases, and document databases.

16.1 Relational Databases

A *relational database* breaks data down into objects and their relationships with one another. A particular entry—like a book—is represented as a row in a table, with columns containing data about books. These columns might include a title, ISBN, and subject. The key thing to remember about relational databases is that different data types reside in different tables.

While one column in a `book` table could be an author's name, it's more likely you'll have an entirely separate `author` table. This table would contain the author's name, perhaps their biography, and an email address. Both tables would then have separate `ID` columns, and the `book` table

might have an `author_id` column referencing the `author` table. Figure 16-1 depicts the relations between tables in this form of database.

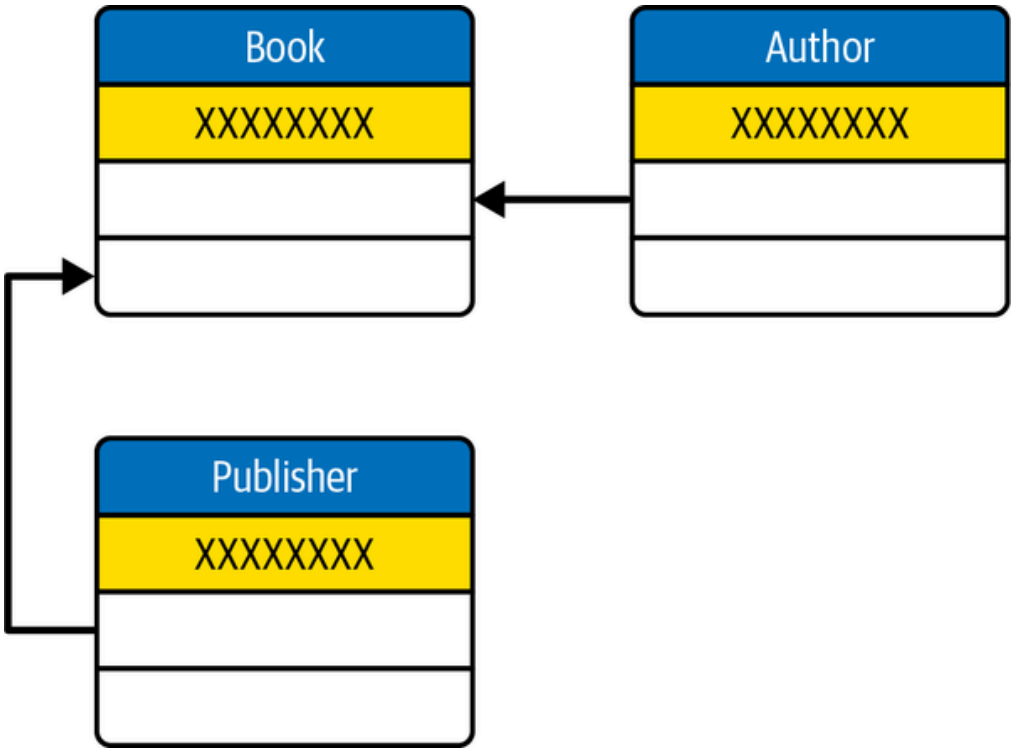

Figure 16-1. Relational databases are defined by tables and references between the items in each

Examples of relational databases include MySQL and SQLite.

16.2 Key-Value Stores

A *key-value store* is far simpler than a relational database—it's effectively a single table that maps one identifier (the key) to some stored value. Many applications leverage key-value stores as simple cache utilities, keeping track of primitive values in an efficient, often in-memory lookup system.

As in relational databases, the data stored in a key-value system can be typed. If you're working with numeric data, most key-value systems expose additional functionality to manipulate that data directly—for example, you can increment integer values without needing to first read the underlying data. Figure 16-2 demonstrates the one-to-one relationships between keys and values in such a data store.

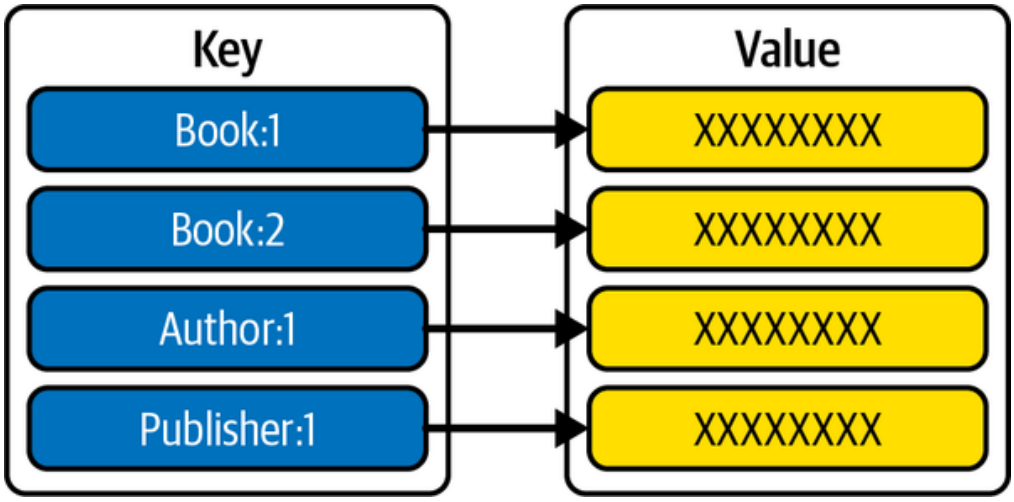

Figure 16-2. Key-value stores are structured as lookups between discrete identifiers mapped to optionally typed values

Examples of key-value stores include Redis and Amazon DynamoDB.

16.3 Graph Databases

Rather than focusing on structuring the data itself, graph databases focus on modeling the relationships (called *edges*) between data. Data elements are encapsulated by nodes, and the edges between the nodes link them together and provide semantic context about the data in the system.

Because of the high priority placed on relationships between data, graph databases are well-suited for visualizations like Figure 16-3, illustrating the edges and nodes within such a structure. They also provide highly efficient queries on data relationships, making them solid choices for highly interconnected data.

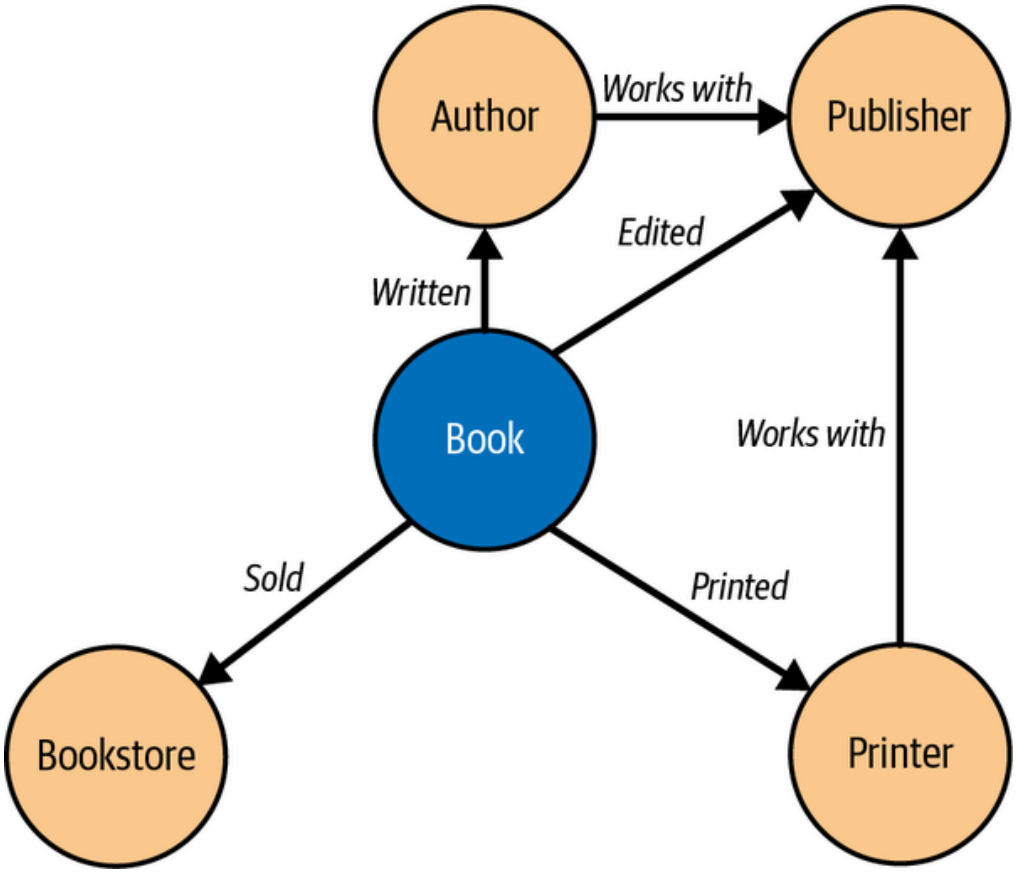

Figure 16-3. Graph databases prioritize and illustrate the relationships (edges) between data (nodes)

Examples of graph databases include Neo4j and Amazon Neptune.

16.4 Document Databases

It's also possible to store data specifically as an unstructured or semistructured *document*. A document could be a well-structured piece of data (like a literal XML document) or a free-form blob of bytes (like a PDF).

The key difference between a document store and the other database types covered in this chapter is structure—*document stores* are typically unstructured and leverage a dynamic schema to reference data. They're incredibly useful in some situations, but far more nuanced in their use. For a deep dive into the document-based methodology, read *MongoDB: The Definitive Guide* by Shannon Bradshaw et al. (O'Reilly).

The following recipes focus primarily on relational databases and how to use them with PHP. You'll learn how to connect to both local and remote databases, how to leverage fixed data during testing, and even how to use a more sophisticated object-relational mapping (ORM) library with your data.

16.5 Connecting to an SQLite Database

Problem

You want to use a local copy of an SQLite database to store application data. Your application needs to open and close the database appropriately.

Solution

Open and close the database as needed using the base `SQLite` class. For efficiency, you can extend the base class with your own constructor and destructor as follows:

```
class Database extends SQLite3
{
    public function __construct(string $databasePath)
    {
        $this->open($databasePath);
    }

    public function __destruct()
    {
        $this->close();
    }
}
```

Then use your new class to open a database, run some queries, and automatically close the connection when you're finished. For example:

```
$db = new Database('example.sqlite');

$create_query = <<<SQL
CREATE TABLE IF NOT EXISTS users (
    user_id INTEGER PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
);
SQL;
```

```
$db->exec($create_query);

$insert_query = <<<SQL
INSERT INTO users (first_name, last_name, email)
VALUES ('Eric', 'Mann', 'eric@phpcookbook.local')
ON CONFLICT(email) DO NOTHING;
SQL;

$db->exec($insert_query);

$results = $db->query('SELECT * from users;');
while ($row = $results->fetchArray()) {
    var_dump($row);
}
```

Discussion

SQLite is a fast, entirely self-contained database engine that stores all of its data in a single file on disk. PHP ships with an extension (enabled by default in most distributions) that directly interfaces with this database, giving you the power to create, write to, and read from databases at will.

The `open()` method will, by default, create a database file if one does not already exist at the specified path. This behavior can be changed by changing the flags passed in as the second parameter of the method call. By default, PHP will pass `SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE`, which will open the database for reading *and* writing as well as create it if it doesn't already exist.

Three flags are available, as listed in Table 16-1.

Table 16-1. Optional flags available for opening an SQLite database

Flag	Description
`SQLITE3_OPEN_READONLY`	Open a database exclusively for reading
`SQLITE3_OPEN_READWRITE`	Open a database for both reading and writing

Flag	Description
`SQLITE3_OPEN_CREATE`	Create the database if it does not exist

The Solution example includes a class that transparently opens an SQLite database at a particular path, creating one if it doesn't already exist. Given that the class extends the base SQLite class, you can then use it in place of a standard SQLite instance to create tables, insert data, and query that data directly. The class destructor automatically closes the database connection once the instance moves out of scope.

NOTE

Typically, closing an SQLite connection isn't explicitly required, as PHP will automatically close the connection when the program exits. If, however, there's a chance that the application (or thread) might continue running, it's a good idea to close your connection to free up system resources as you go. While this won't impact a local, file-based data connection that much, it's a critical component of working with remote relational databases like MySQL. Being consistent in your database management is a good habit to build.

The SQLite database is represented by a binary file on disk at the path specified. If you have a development environment like Visual Studio Code, you can use purpose-built extensions like SQLite Viewer to connect to and visualize your local database as well. Having more than one way to view the schema and data housed within a database is a quick and effective means to validate that your code is doing what you think it's doing.

See Also

PHP documentation for the SQLite3 database extension.

16.6 Using PDO to Connect to an

External Database Provider

Problem

You want to use PDO as an abstraction layer to connect to and query a remote MySQL database.

Solution

First, define a class extending the core `PDO` definition that handles creating and closing connections as follows:

```
class Database extends PDO
{
    public function __construct($config = 'database.ini
    {
        $settings = parse_ini_file($config, true);

        if (!$settings) {
            throw new RuntimeException("Error reading c
        } else if (!array_key_exists('database', $setti
            throw new RuntimeException("Invalid config:
        }

        $db = $settings['database'];
        $port = $db['port'] ?? 3306;
        $driver = $db['driver'] ?? 'mysql';
        $host = $db['host'] ?? '';
        $schema = $db['schema'] ?? '';
        $username = $db['username'] ?? null;
        $password = $db['password'] ?? null;

        $port = empty($port) ? '' : ";port={$port}";
        $dsn = "{$driver}:host={$host}{$port};dbname={$

        parent::__construct($dsn, $username, $password)
    }
}
```

The configuration file for the preceding class needs to be in INI format. For example:

```
[database]
driver = mysql
host = 127.0.0.1
port = 3306
schema = cookbook
username = root
password = toor
```

Once the file is configured, you can query the database directly by using the abstractions provided by PDO as follows:

```
$db = new Database();

$create_query = <<<SQL
CREATE TABLE IF NOT EXISTS users (
    user_id int NOT NULL AUTO_INCREMENT,
    first_name varchar(255) NOT NULL,
    last_name varchar(255) NOT NULL,
    email varchar(255) NOT NULL UNIQUE,
    PRIMARY KEY (user_id)
);
SQL;

$db->exec($create_query);

$insert_query = <<<SQL
INSERT IGNORE INTO users (first_name, last_name, email)
VALUES ('Eric', 'Mann', 'eric@phpcookbook.local');
SQL;

$db->exec($insert_query);

foreach($db->query('SELECT * from users;') as $row) {
    var_dump($row);
}
```

Discussion

The Solution example leverages the same table structure and data as previously used by Recipe 16.5, except that it uses the MySQL database engine. MySQL is a popular, free, open source database engine maintained by Oracle. According to the maintainers, it powers many popular web

applications, including large-scale platforms like Facebook, Netflix, and Uber. In fact, MySQL is so prevalent that many system maintainers ship the MySQL extension with PHP by default, making it even easier to connect to the system and saving you from having to install new drivers by yourself.

NOTE

Unlike the Solution example from Recipe 16.5, PHP has no method to explicitly close the connection when using PDO. Instead, set the value of your database handle (`$db` in the Solution example) to `null` to take the object out of scope and trigger PHP to close the connection.

In the Solution example, you first defined a class to wrap PDO itself and abstract the connection to a MySQL database. This isn't required, but as with Recipe 16.5 it's is a good way to get used to maintaining clean data connections. Once the connection is established, you can create a table, insert data, and read that data back efficiently.

WARNING

The Solution example assumes the `cookbook` schema already existed within the database to which you were connecting. Unless you've already created that schema directly, this implicit connection will fail with a `PDOException` complaining about an unknown database. It is critical that you create the schema *first* within the MySQL database before you try to manipulate it.

Unlike SQLite, MySQL databases require a totally separate application to house the database and broker the connection to your application. Often this application will run on an entirely different server, and your application will connect over TCP on a specific port (usually 3306). For local development and testing, it's enough to stand up a database alongside your application by using Docker. The following one-line command will create a local MySQL database within a Docker container, listening on the default port of 3306 and allowing connections by a `root` user with the password of `toor` :

```
$ docker run --name db -e MYSQL_ROOT_PASSWORD=toor -p 0
```

NOTE

Whether using MySQL within Docker locally or in a production environment, the official container image details various configuration settings that can be used to customize and secure the environment.

When the container first starts, it will not have any schemas available to query (meaning the rest of the Solution example is not yet usable). To create a default `cookbook` schema, you need to connect to the database and create the schema. In Example 16-1, the `$` character indicates shell commands, and the `mysql>` prompt indicates a command run within the database itself.

Example 16-1. Using the MySQL CLI to create a database schema

```
$ mysql --host 127.0.0.1 --user root --password=toor
                                                    ❶


mysql> create database `cookbook`;
                                                    ❷

mysql> exit
                                                    ❸
```

❶ The Docker container is exposing MySQL over TCP to the local environment, which requires you to specify a local host by IP address. Failing to do so defaults to MySQL attempting to connect over a Unix socket, which will fail in this case. You must also pass both the username and password in order to connect.

❷ Once connected to the database engine, you can create a new schema within it.

❸ To disconnect from MySQL, merely type `exit` or `quit` and press the Enter key.

If you don't have the MySQL command line installed, you can also leverage Docker to connect to the running database container and use *its* command-line interface instead. Example 16-2 illustrates how to leverage a Docker container to wrap the MySQL CLI while creating a database schema.

Example 16-2. Using a Docker-hosted MySQL CLI to create a database

schema

```
$ docker exec -it db bash
                                ❶

$ mysql --user root --password=toor
                                ❷

mysql> create database `cookbook`;
                                ❸

mysql> exit

$ exit
                                ❹
```

❶ Since MySQL is already running locally as a container named `db`, you can execute a command within the container interactively by referencing the same name. Docker's `i` and `t` flags indicate you want to execute a command in an interactive terminal session. The `bash` command is what you explicitly want to execute; the result is that you are given an interactive terminal session *within the container* as if you'd connected to it directly.

❷ Connecting to the database within the container is as simple as using the MySQL CLI. You don't need to reference a hostname as, within the container, you can connect directly to the exposed Unix socket.

❸ Creating a table and exiting out of the MySQL CLI is exactly the same as in the previous example.

❹ Once you've exited out of the CLI, you still need to exit out of the interactive `bash` session within the Docker container to return to your main terminal.

There are two primary advantages of using PDO to connect to a database instead of a direct functional interface to the drivers:

1. The PDO interfaces are the same for every database technology. While you might need to refactor specific queries to fit one database engine or another (compare the `CREATE TABLE` syntax of this Solution to that in Recipe 16.5), you don't need to refactor the PHP code around connections, statement executions, or query processing. PDO is a data-access abstraction layer, giving you the same mode of access and

management regardless of the database you happen to be using within your application.

2. PDO supports the use of *persistent connections* by passing a truthy value to the `PDO::ATTR_PERSISTENT` key as an option when opening a connection. A persistent connection will be opened *and remain open* even after the `PDO` instance goes out of scope and your script finishes executing. When PHP attempts to reopen the connection, the system will instead look for a preexisting connection and reuse that if it exists. This helps improve the performance of long-running, multitenant applications, where opening multiple, redundant connections would otherwise harm the database itself. (For more on persistent database connections, review the PHP Manual's comprehensive documentation.)

Beyond these two advantages, PDO also supports the concept of prepared statements, which help reduce the risk of malicious SQL injection. For more on prepared statements, review Recipe 16.7.

See Also

Full documentation on the PDO extension.

16.7 Sanitizing User Input for a Database Query

Problem

You want to pass user input into a database query but don't fully trust the user input to not be malicious.

Solution

Leverage prepared statements in PDO to automatically sanitize user input before it passes into the query as follows:

```
$db = new Database();

$insert_query = <<<SQL
```

```
INSERT IGNORE INTO users (first_name, last_name, email)
VALUES (:first_name, :last_name, :email);
SQL;

$statement = $db->prepare($insert_query);

$statement->execute([
    'first_name' => $_POST['first'],
    'last_name'  => $_POST['last'],
    'email'      => $_POST['email']
]);

foreach($db->query('SELECT * from users;') as $row) {
    var_dump($row);
}
```

Discussion

The concept of sanitizing user input was discussed earlier as part of Recipe 9.1, which used explicit filters to sanitize/validate potentially untrusted input. While that approach is quite effective, it's also easy for developers to forget to include a sanitization filter on user input down the road when making updates. As a result, it's far safer to explicitly prepare queries for execution to prevent malicious SQL injection.

Consider a query used to look up user data to display profile information. Such a query might leverage user email addresses as indexes to distinguish one user from another in an attempt to only display the current user's information. For example:

```
SELECT * FROM users WHERE email = ?;
```

In PHP, you'll want to pass in the current user's email address so the query operates effectively. A naive approach using PDO might look something like Example 16-3.

Example 16-3. Simple query with string interpolation

```
$db = new Database();

$statement = "SELECT * FROM users WHERE email = '{$_POS
```

```
$results = $db->query($statement);
var_dump($results);
```

If the user is only submitting their own username (say, `eric@phpcookbook.local`), then this query will return the appropriate data for that user. There's no guarantee the end user is trustworthy, though, and they might submit a malicious statement instead in the hopes of *injecting* an arbitrary statement into your database engine. Knowing how the submitted email address is interpolated into the SQL statement, an attacker could submit `' OR 1=1;--` instead.

This string will complete the quotes (`WHERE email = ''`), add a composite Boolean statement that matches *any* result (`OR 1=1`), and explicitly comment out any additional characters that follow. The result is that your query will return the data for *all* users rather than the single user who made the request.

Similarly, malicious users could use the same approach to inject arbitrary `INSERT` statements (writing new data) where you expected only to read information. They could also illicitly update existing data, delete fields, or otherwise corrupt the reliability of your data store.

SQL injection is incredibly dangerous. It's also remarkably common in the software world—so much so that injection is recognized as the third most commonly encountered application security risk by the Open Worldwide Application Security Project (OWASP) Top Ten project.

Luckily, in PHP, injection is also easy to thwart!

The Solution example introduces PDO's *prepared statements* interface. Rather than interpolating a string with user-provided data, you insert named placeholders into the query. These placeholders should be prefixed with a single colon and can be any valid name you can imagine. When the query is run against the database, PDO will replace these placeholders with literal values passed in at runtime.

NOTE

It is also possible to use the question mark character as a placeholder and pass values into the prepared statement based on their position within a simple array. However, the position of elements is easy to confuse during later refactoring, and using this simpler approach is highly inadvisable. Take care to always use named parameters when preparing statements to avoid confusion and to future-proof your code.

Prepared statements work with both data manipulation statements (insert, update, delete) and arbitrary queries. Using prepared statements, the simple query from Example 16-3 could be rewritten as Example 16-4.

Example 16-4. Simple query with prepared statements

```
$db = new Database();

$query = "SELECT * FROM users WHERE email = :email;";
$statement = $db->prepare($query);

$statement->execute(['email' => $_POST['email']]);

$results = $statement->fetch();
var_dump($results);
```

This code leverages PDO to automatically escape user input and pass the value as a literal one to the database engine. If the user had in fact submitted their email address, the query would function as expected and return the anticipated result.

If the user instead submitted a malicious payload (e.g., `' OR 1=1;--`, as previously discussed), the statement preparation will explicitly escape the passed quote characters before passing them to the database. This would have the result of looking for an email address that exactly matches the malicious payload (and does not exist), yielding zero results of user data.

See Also

Documentation on PDO's `prepare()` method.

16.8 Mocking Data for Integration Testing with a Database

Problem

You want to leverage a database for production storage but mock that database interface when running automated tests against your application.

Solution

Use the repository pattern as an abstraction between your business logic and database persistence. For example, define a repository interface as shown in Example 16-5.

Example 16-5. Data repository interface definition

```
interface BookRepository
{
    public function getById(int $bookId): Book;
    public function list(): array;
    public function add(Book $book): Book;
    public function delete(Book $book): void;
    public function save(Book $book): Book;
}
```

Then, use the preceding interface to define a concrete database implementation (leveraging something like PDO). Use the same interface to define a mock implementation that returns predictable, static data rather than live data from a remote system. See Example 16-6.

Example 16-6. Repository interface implementation with mock data

```
class MockRepository implements BookRepository
{
    private array $books;

    public function __construct()
    {
        $this->books = [
            new Book(id: 0),
```

```
                new Book(id: 1),
                new Book(id: 2)
            ];
        }

        public function getById(int $bookId): Book
        {
            return $this->books[$bookId];
        }

        public function list(): array
        {
            return $this->books;
        }

        public function add(Book $book): Book
        {
            $book->id = end(array_keys($this->books)) + 1;
            $this->books[] = $book;

            return $book;
        }

        public function delete(Book $book): void
        {
            unset($this->books[$book->id]);
        }

        public function save(Book $book): Book
        {
            $this->books[$book->id] = $book;
        }
    }
```

Discussion

The Solution example introduces a simple way to separate your business logic from your data layer via an abstraction. By leveraging a data *repository* to wrap the database layer, you can ship multiple implementations of the same interface. In a production application, your actual repository might look something like Example 16-7.

Example 16-7. Concrete database implementation of a repository

interface

```
class DatabaseRepository implements BookRepository
{
    private PDO $dbh;

    public function __construct($config = 'database.ini
    {
        $settings = parse_ini_file($config, true);

        if (!$settings) {
            throw new RuntimeException("Error reading c
        } else if (!array_key_exists('database', $setti
            throw new RuntimeException("Invalid config:
        }

        $db = $settings['database'];
        $port = $db['port'] ?? 3306;
        $driver = $db['driver'] ?? 'mysql';
        $host = $db['host'] ?? '';
        $schema = $db['schema'] ?? '';
        $username = $db['username'] ?? null;
        $password = $db['password'] ?? null;

        $port = empty($port) ? '' : ";port={$port}";
        $dsn = "{$driver}:host={$host}{$port};dbname={$

        $this->dbh = new PDO($dsn, $username, $passworc
    }

    public function getById(int $bookId): Book
    {
        $query = 'Select * from books where id = :id;';

        $statement = $this->dbh->prepare($query);
        $statement->execute(['id' => $bookId]);

        $record = $statement->fetch();
        if ($record) {
            return Book::fromRecord($record);
        }

        throw new Exception('Book not found');
    }
```

```php
public function list(): array
{
    $books = [];

    $records = $this->dbh->query('select * from boo
    foreach($record as $book) {
        $books[] = Book::fromRecord($book);
    }

    return $books;
}

public function add(Book $book): Book
{
    $query = 'insert into books (title, author) val

    $this->dbh->beginTransaction();
    $statement = $this->dbh->prepare($query);
    $statement->execute([
        'title'  => $book->title,
        'author' => $book->author,
    ]);
    $this->dbh->commit();

    $book->id = $this->dbh->lastInsertId();

    return $book;
}

public function delete(Book $book): void
{
    $query = 'delete from books where id = :id';

    $this->dbh->beginTransaction();
    $statement = $this->dbh->prepare($query);
    $statement->execute(['id' => $book->id]);
    $this->dbh->commit();
}

public function save(Book $book): Book
{
    $query =
        'update books set title = :title, author =

    $this->dbh->beginTransaction();
    $statement = $this->dbh->prepare($query);
```

```
            $statement->execute([
                'title' => $book->title,
                'author' => $book->author,
                'id' => $book->id
            ]);
            $this->dbh->commit();

            return $book;
        }
    }
```

Example 16-7 implements the same interface as the mock repository from the Solution example, except it connects to a live MySQL database and manipulates data in that separate system. In reality, your production code will use *this* implementation rather than the mock instance. But when running under test, you can easily swap the `DatabaseRepository` for a `MockRepository` instance so long as your business logic is expecting a class that implements `BookRepository`.

Assume you're working with the Symfony framework. Your application will be built upon controllers that leverage dependency injection to handle external integrations. For a library API that manages multiple books, you might define a `BookController` that looks something like the following:

```
class BookController extends AbstractController
{
    #[Route('/book/{id}', name: 'book_show')]
    public function show(int $id, BookRepository $repo)
    {
        $book = $repo->getById($id);

        // ...
    }
}
```

The beauty of the preceding code is that the controller doesn't care whether you pass it an instance of `MockRepository` or one of `DataRepository`. Both classes implement the same `BookRepository` interface and expose a `getByID()` method with the same signature. To your business logic, the functionality is identical—except that with one, your application will reach

out to a remote database to retrieve (and potentially) manipulate data, while the other uses a static, completely deterministic set of fake data.

NOTE

The default data abstraction layer that ships with Symfony is called Doctrine and leverages the repository pattern by default. Doctrine provides a rich abstraction layer across multiple SQL dialects, including MySQL, without the need to manually wire queries via PDO. It also ships with a command-line utility that automatically writes the PHP code for both stored objects (called *entities*) and repositories for you!

When it comes to writing tests, the deterministic and fake data is superior because it will always be the same and means your tests will be very reliable. It also means you won't accidentally overwrite data in a real database if someone makes a minor configuration error locally.

An added advantage will be the speed at which your tests run. Mocked data interfaces remove the need to send data between your application and an independent database, significantly shortening the latency of any data-related function calls. That said, you will likely still want to flesh out a separate integration test suite to exercise those remote integrations, and you will require a real database to make that separate test suite usable.

See Also

Review Recipe 8.7 for more on classes, interfaces, and inheritance. See the Symfony documentation for more on controllers and dependency injection.

16.9 Querying an SQL Database with the Eloquent ORM

Problem

You want to manage your database schema and the data it contains without hand-writing SQL.

Solution

Use Laravel's default ORM, Eloquent, to define your data objects and schema dynamically, as shown in Example 16-8.

Example 16-8. Table definition for use with Laravel

```
Schema::create('books', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('author');
});
```

This code can be used to dynamically create a table to house books, regardless of the type of SQL used with Eloquent. Once the table exists, data within it can be modeled by Eloquent using the following class Example 16-9.

Example 16-9. Eloquent model definition

```
use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    use HasFactory;

    public $timestamps = false;
}
```

Discussion

The Doctrine ORM, mentioned briefly in Recipe 16.8, leverages the repository pattern to map objects stored in a database to their representation in business logic. This works well with the Symfony framework, but is merely one approach to modeling data in a real-world application.

The open source Laravel framework, which itself is built atop Symfony and other components, instead uses the Eloquent ORM to model data. Unlike Doctrine, Eloquent is based on the active record design pattern in which tables within the database are directly related to corresponding models used to represent that table. Rather than creating/reading/updating/deleting models

through a separate repository, the modeled objects present their own methods for direct manipulation.

TIP

Some development teams can be quite opinionated about the design patterns they do and do not accept in a project. Despite the popularity of the Laravel framework, many developers consider the active record approach to data modeling to be an *antipattern*—that is, an approach to be avoided. Take care to ensure that your development team is on the same page regarding the abstractions you leverage in your project, as mixing multiple data access patterns can be confusing and will lead to serious maintenance woes down the line.

The model classes exposed by Eloquent are quite simple, as demonstrated by the terse illustration in the Solution example. However, they are quite dynamic—the actual properties of the model don't need to be directly defined within the model class itself. Instead, Eloquent automatically reads and parses any columns and data types from the underlying table and adds these as properties to the model class when it's instantiated.

The table in Example 16-8, for example, defines three columns:

- An integer ID
- A string title
- A string author name

When Eloquent reads this data directly, it effectively creates objects in PHP that look something like the following:

```
class Book
{
    public int    $id;
    public string $title;
    public string $author;
}
```

The *actual* class will present various additional methods, like `save()`, but otherwise contains a direct representation of the data as it appears within your SQL table. To create a new record in the database, rather than editing SQL

directly, you would merely create a new object and save it as shown in Example 16-10.

Example 16-10. Creating a database object with Eloquent

```
$book = new Book;
$book->title = 'PHP Cookbook';
$book->author = 'Eric Mann';

$book->save();
```

Updating data is similarly simple: use Eloquent to retrieve the object you wish to change, make your changes in PHP, and then invoke the object's `save()` method to persist your updates directly. Example 16-11 updates objects in a database to replace one value in a particular field with another.

Example 16-11. Updating an element in place with Eloquent

```
Book::where('author', 'Eric Mann')
    ->update(['author', 'Eric A Mann']);
```

The key advantage of using Eloquent is that you can work with your data objects as if they were native PHP objects without needing to write, manage, or maintain SQL statements by hand. The even more powerful feature of an ORM is that it handles escaping user input for you, meaning that the extra steps introduced in Recipe 16.7 are no longer necessary.

Although directly leveraging SQL connections (with or without PDO) is a quick and effective way to start working with a database, the sheer power of a fully featured ORM will make your application easier to work with. This is true both in terms of initial development and when it comes time to refactor.

See Also

Documentation on Eloquent ORM.