www.phparch.com

August 2020
Volume 19 - Issue 8

php[architect]

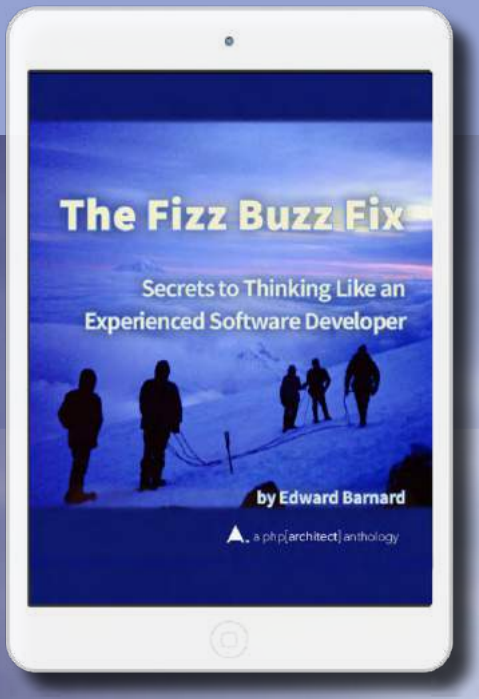
Data Discipline

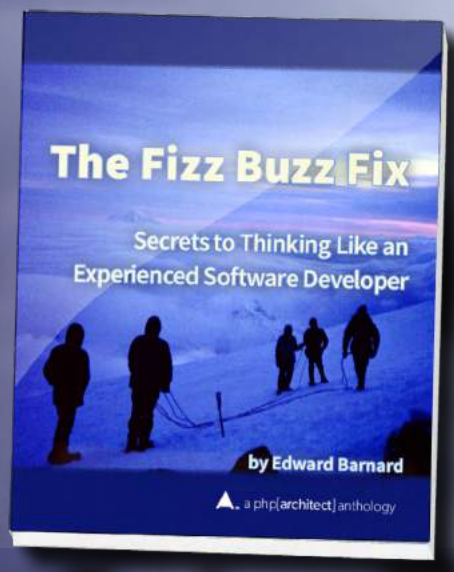
Querying NoSQL With SQL: HAVING Your JSON Cake and SELECTing It Too!

JSON Schema Validation With MySQL

PHP and Database Access

ALSO INSIDE





Tackle Any Coding Challenge With Confidence

Companies routinely incorporate coding challenges when screening and hiring new developers. This book teaches the skills and mental processes these challenges target. You won't just learn "how to learn," you'll learn how to think like a computer. These principles are the bedrock of computing and have withstood the test of time.

Coding challenges are problematic but routinely used to screen candidates for software development jobs. This book discusses the historical roots of why they select for a specific kind of programmer. If your next interview includes a coding exercise, this book can help you prepare.

Available in Print, Kindle Unlimited, and Kindle Lending Library

Order Your Copy

http://phpa.me/fizzbuzz-book

PHP and Database Access

Erwin Earley

In its early days, the Internet was all about sharing static data like images, flat text files, and downloadable binaries. The early Internet's static nature was useful in for sharing information. In many ways, the ability to tie static web pages to the dynamic content resident in enterprise databases allowed us to build new kinds of applications. This is where, in many respects, PHP shines.

Often referred to as the "glue" that binds the static nature of HTML to the dynamic nature of data, PHP has been instrumental in tens-of-thousands of dynamic web applications, including all of those applications based on the ubiquitous LAMP-stack. With that as background, if one is going to work with PHP, one needs to understand both the capabilities that PHP can afford to work with databases and the methods available for working with those databases.

This article takes a somewhat unique approach to explore how PHP works with databases. The first part of the article looks at using PHP to work with the IBM Db2 Database Management System that is integral to the IBM i operating system. After establishing the foundational information, the article delves into PHP's ability to work with the open-source MySQL/MariaDB DBMS. Finally, it will examine how PHP can be used to join data from both DBMSs together programmatically.

PHP on IBM i has been available since 2006, and like other platforms that PHP runs in, it is used for both web-based applications as well as CLI solutions. In the web-space, we see users use PHP as part of the LAMP-stack (yes, MySQL/MariaDB is also available on IBM i) for implementation of solutions based on WordPress, Joomla, and Magento—just to name a few. Additionally, customers use PHP to develop custom applications that leverage their enterprise-level Db2 data and give it a web presence to achieve greater visibility with their constituency and increase the effectiveness and value of their data. A uniqueness of PHP on IBM i is the ability to integrate PHP with existing programs and commands on the system in a way that allows users to leverage existing business logic rather than re-writing that logic. For example, through a capability known as XMLService—part of the IBM i operating system—a PHP program can make a call into an RPG program and then render the results on a web form.

By way of a brief introduction, Db2 on IBM i is an SQL compliant transaction processing Database Management System (DBMS) implemented below the operating system level. This separation is referred to as the Machine Interface that lies between the Operating System and the hardware. The DBMS supports features such as journaling and commitment control such that the data never becomes corrupted, and data isn't lost even if the event of power outages to the underlying hardware. The IBM i operating system is implemented as a database machine in which all objects in the operating systems are, in fact, elements in the Db2 database. The DBMS is exceptionally resilient as well as self-managed. It is unusual to have Database Administrators (DBAs) managing an IBM i system/database. Built on IBM's POWER hardware IBM i scales vertically, and one system can handle multiple, large, and diverse workloads. Total Cost of Ownership (TCO) studies tend to show a competitive cost for IBM i with it's reduced operator expenses and smaller physical footprint without a need for horizontal scaling.

While PHP supports Object Oriented programming as well as methods for data abstraction, this article will use procedural code for its examples along with native drivers for the database access.

IBM Db2

Before getting into the specifics of accessing IBM Db2 from PHP, it's essential to discuss a few features, or nuances of the IBM i operating system. IBM i is essentially a database-machine in that everything in the operating system is, in fact, a database object. As such, connections are to the system rather than a specific database, and libraries are used to segment different database items. For purposes of this article, one could conceptualize a library as a database, and as we will see, libraries can have database tables within them.

As with anything else in PHP, the first place to look is the online documentation for Db2 access[1]. The PHP extension that provides the ability to work with Db2 is ibm_db2. The extension offers upwards of 40 different function calls including those for working with database connections, fetching data, working with field information, working with errors, working with key information working with columns and procedures and working with table information, working with commits and rollbacks, and working with table information. This article will concentrate on the basics of establishing a connection and retrieving data.

The first function to look at is connecting to the database:

```
$conn = db2_connect('127.0.0.1', '', '')
```

The `db2_connect()` function takes four arguments, the system being connected to, the username and password credentials for the connection, and an optional fourth parameter related to the library to scope the connection to. Since PHP

1 *Db2 access: http://php.net/book.ibm-db2*

can run natively on the platform, the connection is often to the localhost, which can be designated as LOCAL or 127.0.0.1. The default configuration for the db2_ibm extension is to allow for a null username and password. In that case, the privileges for the connection will be limited to the privileges of the user running the PHP Apache module which is typically limited to read-only across a limited set of libraries. The limitation of the 'default' user is a security consideration. When user/password credentials are provided on the db2_connect() call, the privileges available via that connection are based on that user credential. Then, database functions such as inserts and updates are supported provided that the user has read/write access to the data source.

As one would expect, the db2_connect() function either returns a resource-handle for the data source, which is stored in the $conn variable or returns FALSE. Error conditions from the connection can be handled with the db2_conn_error() and db2_conn_error_msg() functions, the former returning an error number and the later returning an error string. Connections established with db2_connect() are non-persistent connections that will be torn down when it is explicitly closed using db2_close(). If the script ends without closing the connection, then PHP's cleanup at script termination will close the connection. The ibm_db2 driver also supports persistent connections, which can be established with db2_pconnect(). A connection established with db2_pconnect() is persistent and remains active. When subsequent connections are established for the same user credentials, the connection does not have to be re-established, reducing overhead. In the case of a persistent connection, the connection remains active until db2_pclose() is invoked. Like any other PHP extension, there are numerous settings for ibm_db2 in the ibm_db2.ini file. Two to make a quick mention of are the ability to enforce providing user credentials (ibm_db2_blank_userid) on the connect and treating all connections as persistent connections (i5_all_pconnect). As one might expect, a 0 disables those settings while a 1 enables them.

There are a couple of different ways to retrieve data from IBM Db2 via PHP, and the method to use depends on whether data elements are part of the query. The two functions are db2_exec() which executes an SQL statement directly against the data source and db2_execute() which executes what is referred to as a "prepared" SQL statement to avoid SQL injection attacks. For simplicity, this article will deal exclusively with the db2_exec() function; however, a sidebar to this article will cover the importance of avoiding SQL injection attacks with db_execute() and its related functions. Consider the following two lines of code:

```
$sql = "select * from sales.sp_cust";
$stmt = db2_exec($conn, $sql);
```

The first line establishes a variable with the SQL statement to be executed. Note that the table that the data is being selected from is provided as sales.sp_cust—the sales portion represents the library containing the table. Additional ways to specify the library would have been to provide a single-library or list of libraries as an argument to the connection or to rely on the library-list associated with the user making the connection. From a logical perspective a library-list can be thought of as an execution search path for data sources. db2_exec() attempts to execute the SQL statement ($sql) against the connection resource ($conn) returned from the earlier call to db2_connect(). Successful execution of the SQL statement will result in a statement resource being returned otherwise if the call fails then FALSE is returned. db2_stmt_error() and db2_stmt_errormsg() can be used to work with the error number and error message respectively for a failed db2_connect() or db2_pconnect() call.

Several functions are provided for working with data retrieved from the db2_exec() call and the one to use depends on how the data is to be processed. Consider the following code segment:

```
while ($row=db2_fetch_array($stmt)) {
   list($number, $firstname, $lastname) = $row;
   echo "$number, $firstName, $lastName" . "\<br\>";
}
```

The previous code uses a while loop to process through the set of records returned from the SQL SELECT statement one row at a time. db2_fetch_array() returns an enumerated indexed array. For code readability, list() is used to take the field values and place them in scalar variables so we can use those same scalar variables in the echo statement. There are other functions provided by the ibm_db2 extension for working with a row in a result set:

Function	Description
db_fetch_array()	Returns an array, indexed by column position. Column indexing starts at 0
db2_fetch_assoc()	Returns an array, indexed by column name.
db2_fetch_both()	Returns an array, indexed by column name and position. Keep in mind that the row returned by db2_fetch_both() will require more memory then the single-indexed arrays returned by db2_fetch_assoc() and db2_fetch_array()
db2_fetch_object()	Returns an object in which each property represents a column returned in the row fetched from a result set.

In addition to the usability differences of each retrieval function there are also performance differences that should be considered with db2_fetch_array() being the most performant while db2_fetch_assoc() and db2_fetch_both() add overhead due to the need to retrieve the field names from the data-source. Speaking of field-names, there are a number of functions provided for working with field information including db2_field_name(), db2_field_num(),

db2_field_precision(), db2_field_scale(), db2_field_type(), db2_field_width(), and db2_field_display_size().

There are other functions for working with results from an SQL statement db2_result() returns a single column from a row in the result set, and db2_fetch_row() sets the pointer to the next row or the requested row depending on how the function is called. These are considered second-tier functions, and as such, they are seldom used in favor of the functions mentioned earlier for working with results.

It should be noted that db2_stmt_error() and db2_stmt_errormsg() can be used to work with errors returned from the data retrieval functions.

Keep in mind that while the example shows an SQL SELECT statement, we can execute virtually any SQL statement, including INSERT, UPDATE, and DELETE statements. Together with the SELECT statement form, these are the basis for the essential database activities of any data-driven application, namely CRUD (Create, Read, Update, Delete).

PHP

The next section of the article will look at MySQL/MariaDB access from PHP. For simplicity, the remainder of the article will refer to MySQL; however, one should keep in mind that MariaDB is a drop-in replacement for MySQL. Everything that follows can be used for MariaDB environments as well. The PHP extension that provides support for MySQL is mysqli[2]. It is the MySQL Improved extension that has been available since PHP 5 and is designed to work with version 4.1.3 and later of MySQL and the equivalent version of MariaDB. The mysqli extension has support for procedural and object-oriented coding practices—like the previous discussion of IBM Db2. This article limits itself to the procedural code for working with MySQL.

Connections to the database server are made with the mysqli_connect() function, which takes as it's arguments the hostname where MySQL DBMS is installed and the username and password credentials for the user on the MySQL DBMS. An optional fourth parameter allows for specifying the database to connect to. Unlike the ibm_db2 driver that has a separate function for establishing persistent connections, persistent connections to MySQL are established by prepending a p: to the hostname parameter of the mysqli_connect() call.

Unlike ibm_db2 which provided separate functions for working with errors on database connections vs. SQL processing, MySQL has a single set of functions, specifically mysqli_errorno() and mysqli_error() which returns the error number and error text respectively for working with both connection and SQL processing errors.

mysqli_select_db() is used to choose or change the default database to execute database queries against. Recall that the mysqli_select_db() function has an optional parameter for establishing the default database on connection. It is typical to do so and only use mysqli_select_db() for those times when the default database needs to be changed within the PHP script. The following code provides a connection and database selection example:

```
$con2 = mysqli_connect("host", "user", "password");
mysqli_select_db($con2, "dbname");
```

The return from mysqli_select_db() is either TRUE or FALSE. Normally the call would be in an if statement to test for success or failure of the call and react to errors accordingly

Execution of SQL against the MySQL data source is handled by the mysqli_query() function. Like the db2_exec() function, mysqli_query() executes an SQL statement directly on the DBMS and is subject to SQL injection attacks. One could (and should) use mysqli_stmt_prepare() along with mysqli_stmt_bind_param() and mysqli_stmt_execute() to avoid SQL injection attacks.

Like the ibm_db2 extension, the mysqli extension provides several functions for working with the set of records returned from a query including mysqli_fetch_row() which returns the row as an enumerated array and mysqli_fetch_assoc() which returns the row as an associative array where the keys are the field names. Another function provided for working with the records is mysqli_fetch_array(), which returns the row as either an associative or enumerated array or both based on the result-type requested in the call. After working with the result set, it should be closed with mysqli_free_result().

Connections to MySQL are closed with mysqli_close(). It should be noted that non-persistent MySQL connections and result sets are automatically closed when the PHP script finishes, whereas persistent connections remain open until explicitly closed. That said, it is considered a best practice to close persistent as well as non-persistent connections explicitly.

The following table provides a mapping of ibm_db2 provided functions to their mysqli counterparts:

Description	ibm_db2	mysqli
Establish connection	db2_connect()	mysqli_connect() mysqli_select_db()
Work with connection errors	db2_conn_error() db2_conn_error_msg()	msyqli_errorno() mysqli_error()
Close connection	db2_close()	mysqli_close()
Establish persistent connection	db2_pconnect()	mysqli_connect(p:host…)
Close persistent connection	db2_pclose()	mysqli_close()

2 *mysqli: https://php.net/book.mysqli*

Description	ibm_db2	mysqli
Execute an SQL statement	db2_exec() db2_execute()	mysqli_query() mysqli_stmt_execute()
Work with SQL errors	db2_stmt_error() db2_stmt_errormsg()	mysqli_errorno() mysqli_error()
Retrieve data	db2_fetch_array() db2_fetch_assoc() db2_fetch_both()	mysqli_fetch_row() mysqli_fetch_array() mysqli_fetch_assoc()
SQL injection related	db2_prepare() db2_bind_param() db2_execute()	mysqli_stmt_prepare() mysqli_stmt_bind_param() mysqli_stmt_execute()

Bringing The DBMSs Together

Let's see how these two disparate data sources can be joined together via PHP now that we've established a baseline of knowledge for working with IBM Db2 data and MySQL data. Items to keep in mind is that the return from a successful connection to a DBMS from PHP is a resource, and that resource is used by PHP functions to work with the DBMS. Multiple DBMS connections (i.e., multiple resources) can be active within a PHP application. Any type of "union" function (such as joining across tables) that spans various DBMS (resource) is done via program logic. Consider the code segment in Listing 1

The first two code lines establish connections to the IBM Db2 data source and the MySQL data source. Notice that we store the resource returned from the respective connect calls in separate variables. Additionally, note that we select the database on the connect call for the MySQL connection, so a subsequent `mysqli_select_db()` call isn't needed.

The next two code lines (following the "Select records from Db2" comment) establish and execute the SQL statement that retrieves records from IBM Db2—one could think of this as an outer table join, as shown in a minute.

The first while statement establishes a while loop that will process through the records returned from the SELECT against Db2. The processing within the while loop is where things get interesting. The `$name =` assignment takes the first element from the array representing the current record from the IBM Db2 data retrieval and stores it in a scalar that will be used for a subsequent select from MySQL.

After storing the name value, the next two lines of code establishes and executes the SQL statement that retrieves records from the MySQL data source using the field value saved from the Db2 record, which can be thought of as a foreign-key. The result of this `SELECT`, which can be considered an inner join, is to retrieve MySQL records that have a relationship with the current records from Db2. Once the record is retrieved, we use a do-while loop to iterate through the MySQL result set and output the retrieved records.

Listing 1.

```
1. <?php
2.
3. // Establish database connections
4. $con1 = db2_connect('localhost', '', '');
5. $con2 = mysqli_connect(
6.    'localhost', 'dbuser', 'password', 'testdb'
7. );
8.
9. // Select records from Db2
10. $sql = "select * from sp_cust";
11. $result = db2_exec($con1, $sql);
12.
13. // Process 1st result set
14. while ($row1 = db2_fetch_array($result)) {
15.    $name = $row1[0];
16.    // select from second database
17.    $sql2 = "select * from crm where custname = $name";
18.    $result2 = mysqli_query($con2, $sql2);
19.    // process through return set
20.    $row2 = mysqli_fetch_row($result2);
21.    do {
22.       echo "<tr><td>{$row2[0]}</td>";
23.       echo "<td>{$row2[1]}</td>";
24.       echo "<td>{$row2[2]}</td>";
25.       $row2 = mysqli_fetch_row($result2);
26.    } while ($row2);
27. }
```

Keep in mind that while the example doesn't include error checking, good coding practices would have included error checks at each appropriate point within the code (such as each connection attempt, SQL query execution, and data retrieval functions).

One final item worth noting is that for both IBM i Db2 and MySQL, there are PHP extensions that support PHP Data Objects (PDO), which provides a uniform method of access to multiple databases. The extension that provides PDO for IBM i Db2 is pdo_ibm, while the extension for MySQL is pdo_mysql. Since PDO abstracts data access away from the databases themselves, the functions (or in the case of object-oriented PDO, the methods) become consistent, with only the database connection being unique between the DBMSs.

So as an example, rather than having to use different functions for execution of the query (`db2_exec()` and `myqli_query()` in the example) and different functions for accessing the results (`db2_fetch_array()` and `mysqli_fetch_row()` in the example) the calls become consistent across DBMSs used—`query()` and `fetch()` methods, respectively. Note, however, that the SQL for each DBMS might differ, taking advantage of each vendor's proprietary extensions to SQL and subject to its restrictions. You don't use MySQL's backticks to quote identifiers when working with DB2, regardless of whether you use PDO.

Avoiding SQL Injection

When developing data-centric applications, it is essential to keep security at the forefront of both the application's design and implementation. One way to ensure data security is through the filter extension provided for PHP, which includes many filter types, including "validate" and "sanitize." Check out https://php.net/book.filter for more information on the filter extension.

This sidebar looks at SQL injection attacks and how to avoid them in PHP. SQL injection is exactly what the name implies. It's injecting data/statements into an SQL statement. Consider the following SQL `INSERT` statement:

```
select * from dbtable where customer = $name;
```

Let's further assume that the value for `$name` comes from a web-form (probably a safe assumption since this is likely a PHP application. Without proper hygiene of the web-form or validation of the data, a user could input the following for name:

```
John;truncate sales;
```

This input results in the following SQL statements (yes statements):

```
select * from dbtable where customer = John; truncate sales;
```

When executing this string, besides performing the `select` statement, the records from the `sales` table are deleted. It's highly unlikely that that is the intended result. So how can this be prevented from occurring? One way is with prepared statements. With a prepared statement, instead of sending a raw query (as shown earlier) to the database engine, the database engine is first told the structure of the query that will be submitted.

One way to avoid SQL injection is through the validation of input, as briefly discussed above. Another way to avoid SQL injection is to use a prepared query that uses placeholders for the parameters of the query statement and then binds values to those parameters. Consider this example which uses an SQL INSERT statement:

```
INSERT into dbtable (name) VALUES ($name);
```

At this point, it is still possible to have malicious statements injected via the data represented by the `$name` variable being passed to the database. Using a placeholder changes the statement to the following:

```
INSERT into dbtable (name) VALUES (?);
```

Now, injection isn't possible since no value (variable or literal) is sent to the database engine. The parameterized statement (sometimes referred to as a template) is sent to the database engine with the `db2_prepare()` function.

Listing 2.

```
1. <?php
2.
3. $con = db2_connect('localhost', 'dbuser', 'userpass');
4. $sql = "INSERT into dbtable (name) values(?)";
5. $stmt = db2_prepare($con, $sql);
6. db2_bind_param($stmt, 's', $name);
7. db2_stmt_execute($stmt);
8. db2_close($con);
```

Getting the values themselves to the database is done with the `db2_bind_param()` function, and finally, we execute the statement with the `db2_execute()` function. The example in Listing 2 shows this in practice.

Since the bound variables are sent to the database engine separate from the query, they cannot be interfered with. The database engine uses the values directly at the point of execution after the statement itself has been parsed. The above code should be expanded to included error checking along the way at each database function execution.

While the examples in this side-bar used IBM Db2 as the data source, the same method for avoiding SQL injection can be implemented for MySQL data sources through use of the `mysqli_stmt_prepare()`, `mysqli_stmt_bind_param()`, and `myqli_stmt_execute()` functions.

You should also avoid using user-supplied input for field names, database tables, and sorting or grouping parameters. If you do, make sure you escape them appropriately as these can not be parameterized.

Conclusion

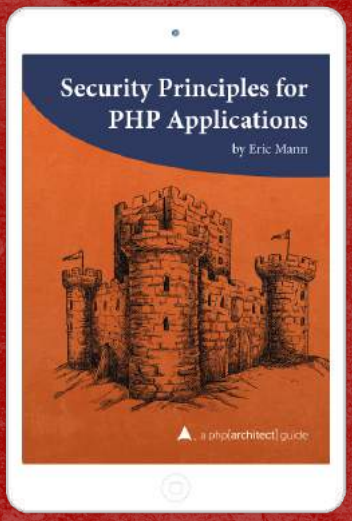
This article set out to explore basic database functions provided by PHP for both IBM Db2 and MySQL and show how data from those disparate DBMSs could be programmatically joined together to provide additional functionality and insight from the logical relationships between the data sources.

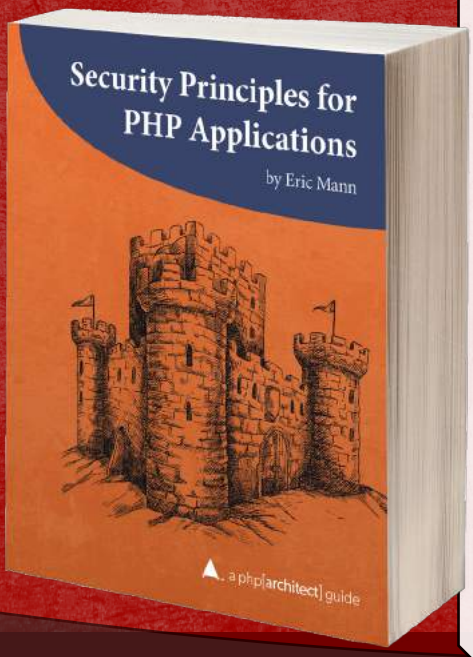
Erwin offers more than 20 years of experience working in the IBM i community, as an IBM employee and now with Zend by Perforce. He has worked with many technologies on the POWER platform including cloud (PowerVC), Dev/Ops solutions (Docker and Chef), and open source technologies (Linux, MySQL/MariaDB, and PHP).

Related Reading

- *Education Station: An Introduction to Doctrine*
  by Matthew Setter, September 2017.
  https://phparch.com/magazine/2017-2/september/
- *PHP Prepared Statements and MySQL Table Design*
  by Edward Barnard, May 2017.
  https://phparch.com/magazine/2017-2/may/
- *Practical Database Design*
  by David Berube, March 2015.
  https://phparch.com/magazine/2015-2/march

a php[architect] guide





Discover how to secure your applications against many of the vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

Security Principles for PHP Applications is a comprehensive guide. This book contains examples of vulnerable code side-by-side with solutions to harden it. Organized around the 2017 OWASP Top Ten list, topics cover include:

- Injection Attacks
- Authentication and Session Management
- Sensitive Data Exposure
- Access Control and Password Handling
- PHP Security Settings
- Cross-Site Scripting
- Logging and Monitoring
- API Protection
- Cross-Site Request Forgery
- ...and more.

Read a Sample Online

Written by PHP professional Eric Mann, this book builds on his experience in building secure, web applications with PHP.

Order Your Copy

http://phpa.me/security-principles

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]