www.phparch.com

April 2018
Volume 17 - Issue 4

Testing in Practice

php[architect]

PHPUnit Worst Practices

Easier Mocking with Mockery

Testing Strategy With the Help of Static Analysis

Evolved PHP

ALSO INSIDE

Artisanal: Authentication with Laravel

The Dev Lead Trenches: Ongoing Education

Community Corner: What's the Fuss About Serverless?

Security Corner: PHP Isolation in Production

Education Station: Build an API, Part One

finally{}: On the Creativity of Programmers

PHP[TEK] 2018

The Premier PHP Conference

13th Annual Edition

MAY 31ST – JUNE 1ST

Downtown Atlanta

Full schedule announced!

Talks from experts at MailChimp, Salesforce, Etsy, Google, Oracle, Pantheon and many more!

tek.phparch.com

CONTENTS

php[architect]

APRIL 2018
Volume 17 - Issue 4

Features

Columns

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

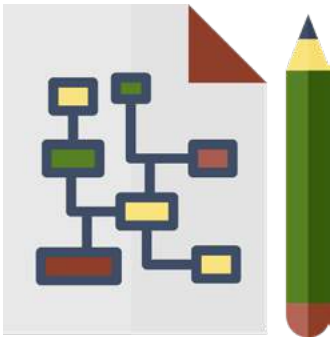
Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Testing Strategy With the Help of Static Analysis



Ondrej Mirtes

When developing an application, our aim as software developers is to make sure it does what it ought to do and to keep the number of defects as low as possible. We should also strive to make our lives easier, to counter external circumstances like tight deadlines and ever-changing requirements causing the exact opposite. That's why we're always looking out for tools and practices to help us with our jobs.

In this article, I'd like to introduce you to the concept of type safety and how it can improve the reliability and stability of your code. Once your code is more type-safe, and that fact is verified by automated tools, you can cherry-pick which parts of your application need extensive unit tests and where you can rely just on well-defined types.

Type System And Type Safety

To have a type system means to communicate what kinds of values travel through code clearly. Since not all values can be treated the same, the more we know about them, the better. If you currently don't have any type hints at all, adding information to the code whether you're accepting `int`, `float`, `string` or `bool` can go a long way.

But when a function declares it accepts an integer, does it really mean any integer? Just a positive integer? Or only a limited set of values, like hours in a day or minutes in an hour? Trimming down possible inputs reduces undefined behavior. Going down this road further means you have to start type-hinting your own objects which comes with additional benefits—not only that we know what we can pass to the function, it also tells us what operations (methods) the object offers.

I'm not saying scalar values are never enough, and you should always use objects, but every time you're tempted to type hint a `string`, go through a mental exercise on what could go wrong with the input. Do I want to allow an empty string? What about non-ASCII characters? Instead of putting validation logic into a function that does something with a scalar value, create a special object and put the validation logic in its constructor. You don't have to write the validation in each place where the object is accepted anymore, and you also don't have to test the function's behavior for invalid inputs provided the object cannot be created with invalid values.

For example, you might have a function which accepts a string for an email address, but you must check if the email is valid.

```
function sendMessage(string $email) {
   if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
      throw new \InvalidArgumentException(
         "Invalid email string"
      );
   }
   // do something
}
```

Instead, you can flip it and make your function—and any other one—explicitly expect an `EmailAddress` object as in Listing 1.

Once your codebase is filled with type hints, IDEs and static analyzers know more about it, and you can benefit from them. For example, if you annotate a property with a phpDoc (unfortunately there's no native support for property types yet), these tools can verify:

Listing 1

```
1. class EmailAddress
2. {
3.     /** @var string */
4.     private $email;
5.
6.     public function __construct(string $email) {
7.        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
8.           throw new \InvalidArgumentException(
9.              "Not a valid email string"
10.          );
11.       }
12.        $this->email = $email;
13.    }
14.
15.    public function getAddress(): string {
16.       return $this->email;
17.    }
18. }
19.
20. function sendMessage(EmailAddress $email) {
21.    // do something
22. }
```

1. The type hint is valid, and the class exists.
2. You're assigning only objects of this type to it.
3. You're calling only methods available on the type-hinted class.

```
/**
 * @var Address
 */
private $address;
```

The same benefits stemming from type hints are also applicable to function/method parameters and return types. There's always the write part (what is returned from a method) and the read part (what the caller does with the returned value).

Listen to the Types

Types can also give you subtle feedback about the design of your application—learn to listen to it. One such case is inappropriate interfaces—when you're implementing an interface, and you're forced to throw an exception from half of the methods you have to add to the class, the interface isn't designed well and will usually benefit from separating into several smaller ones. Using such an interface in production code is dangerous—by implementing it, you're promising it's safe to pass the object somewhere the interface is type-hinted but calling its methods will throw unexpected exceptions.

Another type of feedback is making use of information that's unknown to the type system. If the developer knows and takes advantage of something that isn't obvious from looking at the type hints, like checking a condition in advance and knowing what a method will return based on the result. It can make the tools fail with a false positive:

```
if ($user->hasAddress()) {
    // getAddress() return typehint
    // is ?Address
    $address = $user->getAddress();

    // potentially dangerous -
    // $address might be null
    echo $address->getStreet();
}
```

There's no machine-readable connection between `hasAddress()` and `getAddress()` in type hints. The knowledge the above code will always work is available only in developer's head or by closely inspecting the source code of the class. You might object that this example is too simple, and everyone understands what's going on, but there are much more complex examples like this in the wild. For example, a `Product` object that has every property nullable because they can be empty while the `Product` is being configured in the content management system, but once it's published and available for purchase on the site, they are guaranteed to be filled out. Any code that works only with published products has to make `$value !== null` checks in order to comply with the type system.

A solution to this problem is generally not to reuse objects for different use cases. You can represent published products with a PublishedProduct class where every getter is non-nullable.

Tools for Finding Type System Defects (a.K.a. Bugs)

Because it is interpreted at runtime, PHP itself does not discover type system defects in advance because that's usually a job of the compiler. A program in C# or Java will refuse to execute if there's a problem like an undefined variable, calling an unknown method or passing an argument of a wrong type somewhere deep in the code. In PHP, if there's an error like that in the third step of the checkout process, the developer (or the user) will find it when they execute that line of code during testing or in production. But thanks to the latest advancements in the language itself, like scalar and nullable type hints, it's now easier to be sure about types of many variables and other expressions just by looking at the code without the need to execute it.

That's where static analyzers come into play. They gather all available information about the code—besides native type hints, they understand common phpDoc conventions, employ custom plugins and analyze loops and branches to infer as many types as possible.

One of these tools is PHPStan[1]; it's open-source and free to use (*disclaimer: I'm the main developer of PHPStan.*) Other similar tools are Phan[2], Exakat[3], and Psalm[4].

Besides obvious errors, it can also point out code that can be simplified like always false comparisons using `===`, `!==`, `instanceof`, or `isset()` on never defined variables, duplicate keys in literal arrays, unused constructor parameters, and much more. Because running a comprehensive analysis on an existing codebase for the first time can result in an overwhelming list of potential issues, PHPStan supports gradual checking. Its goal is to enable developers to start using the tool as soon as possible and to feel like they're leveling up in a video game.

```
vendor/bin/phpstan analyse src/
```

If you run the PHPStan executable without any flags, it will run the basic level zero by default, checking only types it's completely sure about, like methods called statically and on `$this`. It does not even check types passed to method arguments until level five (only the number of passed arguments is checked on lower levels), but it definitely finds a lot of issues in between.

PHPStan is extensible—you can write custom rules specific to your codebase and also extensions describing behavior of magic `__call`, `__get`, and `__set` methods. You can also write a so-called "dynamic return type extension" for describing return types of functions or methods which vary based on various conditions like types of arguments passed to the function or the type of object the method is called on. There are already plenty of extensions available for popular frameworks like Doctrine, Symfony, or PHPUnit.

1 *PHPStan:* https://github.com/phpstan/phpstan
2 *Phan:* http://github.com/phan/phan
3 *Exakat:* https://exakat.io
4 *Psalm:* https://getpsalm.org

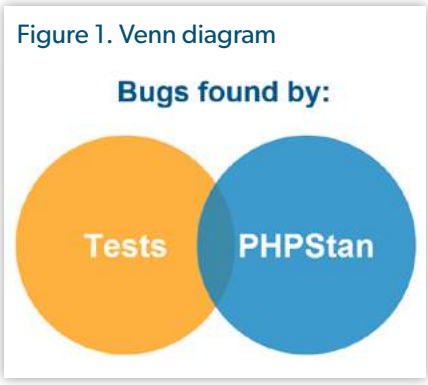
How to Save Time with a Static Analyzer

We already established there is a lot of value to gain from the type system. But how can we use it to save precious time and resources? When testing a PHP application, whether manually or automatically, developers spend a lot of their time discovering mistakes that wouldn't even compile in other languages, leaving less time for testing actual business logic. Typically, there's duplicate effort because some bugs are discovered by both static analysis and by unit tests as in Figure 1.

Since tests must be written by humans and represent code as any other, they are very costly—not only during the initial development but for maintenance as well. Our goal should be to make those two sets disjoint, so we don't write any test which can be covered by static analysis. And while we're at it, we can try to make the blue circle as big as possible so the type system gains more power and is able to find most bugs on its own.

One could object we'll save time by not writing redundant tests, but that time will get used up by writing classes, adding type hints, thinking about interfaces and structuring the code differently to benefit from the type system as much as possible. To counter the objection, I'd argue tests get written only to prevent bugs, but solid and strong types have benefits reaching much further—they improve readability and provide necessary communication and documentation about how the code works. Without any types, the orientation in the code is much harder not only for static analyzers but for developers too.

Figure 1. Venn diagram



PHP is naturally very benevolent about the handling of data which usually goes against safety and predictability. Many of the tips I'm sharing below are about cutting down the space of possible outcomes, resulting in simpler code.

Tips for More Strongly-Typed Code

1. Don't Hide Errors

Turn on reporting and logging of all errors using `error_reporting(E_ALL);`. Especially notices (e.g., `E_NOTICE`), regardless of their name, are the most severe errors that can occur—things like undefined variables or missing array keys are reported as notices.

2. Enable Strict Types

Use `declare(strict_types = 1);` on top of every file. This ensures only values of compatible types can be passed to function arguments, basically that `"dogs"` does not get cast to `0`. I can't recommend this mode enough; its impact can be compared to turning on notice reporting. The per-file basis allows for gradual integration—turn it on in a few selected files and observe the effects, rinse and repeat until it's on in all the files.

3. Encapsulate All Code

All code should be encapsulated in classes or at least functions. Having all the variables created in the local scope helps tremendously with knowing their type. For the same reason, you shouldn't use `global` variables. Instead of procedural scripts stitched together via `include` and using variables appearing out of nowhere, everything is neatly organized and obvious.

4. Avoid Unnecessary Nullables

Avoid nullables where they're not necessary. Nullable parameters and return types complicate things. You have to write `if` statements to prevent manipulating with `null`. More branches signify there's more code to test. Having multiple nullable parameters in a method usually means only some nullability combinations are valid. Consider this example which sets a date range on an object:

```
public function setDates(
   \DateTimeImmutable $from,
   \DateTimeImmutable $to);
```

If a requirement comes that the dates should also be removable, you might be tempted to solve it this way:

```
public function setDates(
   ?\DateTimeImmutable $from,
   ?\DateTimeImmutable $to);
```

But that means you can call the method with only a single date, leaving the object in an inconsistent state.

```
$object->setDates(
   new \DateTimeImmutable('2017-10-11'),
   null);
$object->setDates(
   null,
   new \DateTimeImmutable('2017-12-07')
);
```

You can prevent this from happening by adding an additional check to the method body, but that bypasses the type system and relies only on runtime correctness. Try to distinguish between different use cases not by making the parameters nullable but by adding another method:

```
public function setDates(
   \DateTimeImmutable $from,
   \DateTimeImmutable $to);
public function removeDates();
```

5. Avoid associative arrays.

When type-hinting an `array`, the code does not communicate that the developer should pass an array with a specific structure. And the function which accepts the array cannot rely on the array having specific keys and that the keys are of a certain type. Arrays are not a good fit for passing values around if you want maintainability and a reliable type system. Use objects which enforce and communicate what they contain and do. The only exception where arrays are fine is when they represent a collection of values of the same type. This can be communicated with a phpDoc and is enforceable using static analysis.

```
/**
 * @param User[] $users
 */
public function setUsers(
   array $users
);
```

6. Avoid Dynamic Code

Avoid dynamic code like `$this->$propertyName` or `new $className()`. Also, don't use reflection in your business logic. It's fine if your framework or library uses reflection internally to achieve what you're asking it to do but stay away from it when writing ordinary code. Do not step outside the comforts of the type system into the dangerous territory of reflection.

7. Avoid Loose Comparisons

Don't use loose comparisons like `==` or `!=`. When comparing different types, PHP will try to guess what you mean which leads to very unexpected results. I circled the most surprising ones in Figure 2. Did you know `"0"` equals to `false`? Or that `array()` equals to `null`?

Instead, use strict comparisons like `===` and `!==`. They require both the types and the values to match. In case of objects, `===` will return true only if it's the same instance on both sides. In case of `DateTime` objects, where the comparison operators are overloaded by the language, using `==` was acceptable. When PHP 7.1 introduced microseconds, it broke a lot of code. I recommend comparing `DateTime` instances by first calling `->format()`, stating the required precision and then compare the strings using strict operators.

Avoid Empty Comparisons

A similar case of loose typing is the `empty` construct. Here's a list of values considered empty:

- `""` (an empty string)
- `0` (an integer)
- `0.0` (a float)
- `"0"`
- `null`
- `false`
- an empty array

This makes `empty` unusable for any input validation. Instead, work with specific values and write a specific comparison of what you're trying to achieve. Don't use `empty` when asking about an empty array, use `count($array) === 0`. Don't use it for detecting an empty string because it would not accept `"0"`, write `$str !== ''` instead.

9. Use only Booleans in conditions

If you look at the table summarizing loose comparisons, the `true` and `false` columns summarize what happens when a bare value is put into a condition, and the result can surprise you. Again, use explicit comparison against the value you're looking for. **Tip**: PHPStan's extension `phpstan/phpstan-strict-rules` enforces this and other rules for strictly-typed code.

What Tests Do We Need?

All the tips above were of local character—how to improve specific places in your code to get the most out of the type system. What I'm about to share now impacts the architecture of the whole application and makes opinionated decisions about what has to be unit-tested and where the static analysis of the type system is sufficient.

First, let's clarify why I think a unit test is the most valuable kind of test and what criteria it needs to meet. Unit tests focus on testing one small unit, usually a class or a function. When a unit test fails, we know exactly which place needs to get fixed. In contrast, when an integration or even a UI test fails, we have no idea where to look. A third party service could be down, our database could have changed, or maybe we just moved some button 10 pixels to the right and changed its text. Unit tests shouldn't have any dependencies outside of the code. They should not connect to the database, access the filesystem and send anything over the network. Since they focus on testing just the application code, they tend to be very fast. Having some integration tests help, mainly for things that can't be tested with unit tests because of their nature, but unit tests should form the big and solid base of your test pyramid.

Application code can be divided into two main types: wiring and business logic. Wiring is what holds the application together; controllers, facades, passing values along, getters and setters. Business logic is what justifies the existence of our application and what we're paid to do, such as mathematical operations, filtering, validation, parsers, managing state transitions, etc. A static analyzer will tell you if you're calling an undefined method, but it can't know you should have used `ceil()` instead of `floor()` or that you should have written that `if` condition the other way around.

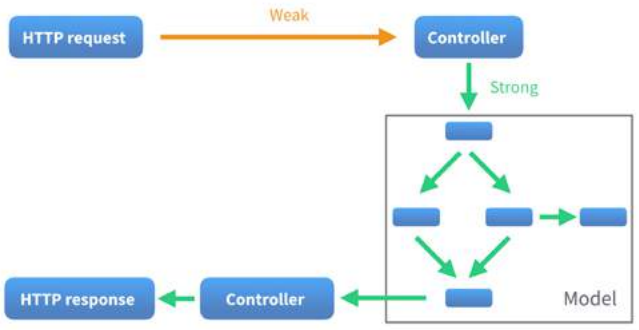
So how can one justify the existence of wiring? In modern applications, there's a lot of it, and we're better for it. Wiring makes reusability possible. We

Figure 2. Loose comparison table

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Figure 3. Grow



Listing 2

```
1. class Calculator
2. {
3.     /**
4.      * @param Foo $foo
5.      * @param Bar[] $bars
6.      */
7.     public function calculate(Foo $foo,
8.                               array $bars): CalculatorResult {
9.         // ...
10.    }
11. }
```

can extract common pieces of code and call them from multiple places. It improves readability by splitting code into smaller chunks. Thanks to wiring, our code is more maintainable.

But because of its nature, testing it can become tedious and boring. Testing setters, getters, and whole classes whose only purpose is to forward values to subsequent layers does not yield a lot of revealed errors and is not very economical. Once we have type hints for everything and employ a static analyzer, it should be enough to verify the wiring code works as expected.

The role of the entry point to an application, like a controller, is to sanitize all incoming data and pass them further down the line as well-defined and validated types. If you allow strong types to grow through your application, it becomes much more solid and statically verifiable as shown in Figure 3.

Unit tests should focus on business logic. A classic mistake is to interweave business logic with data querying, making the job of a unit test much harder:

```
function doSomething(int $id): void {
    $foo = $this->someRepository
        ->getById($id);
    // business logic performed on $foo.
    $bars = $this->otherRepository
        ->getAllByFoo($foo);
    // business logic performed on $foo
    // and $bars
}
```

With this code structure, you don't have any other option than to mock in order to provide the interesting lines you want to test with some data. Mocking in a correctly architectured application shouldn't be necessary because it's white-box testing and by definition dependent on the inner structure of the tested code, therefore more prone to breaking.

Instead, separate the business logic a to a different class and design its public interface to receive all the data it needs for its job. This class does not need to perform any database queries or other side effects. It will receive data from the real source in production and manually created data in tests. You can write a lot of simple unit tests without any mocking and test every edge case since you saved a lot of time by not testing the wiring code (Listing 2)!

This is also the reason why I like using ORMs like Doctrine. They get a bad reputation because people use them for the wrong reasons. You shouldn't look to an ORM to generate SQL queries for you because the tool doesn't know what you will need in advance, resulting in poor performance. You shouldn't use an ORM so you can switch to a different database engine one day. Quite the opposite—you're missing out if you're not using the advanced features of your database of choice. The reason why I like to use an ORM is because it allows me to represent data as objects—they can be type-hinted, constructed by hand for the purpose of unit tests, and contain methods which guarantee consistent modifications.

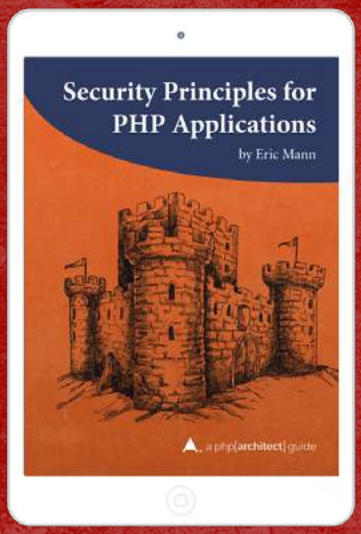
With tests, you can measure code coverage, a percentage of executed lines of code during test runs. I propose a similar metric for static analysis. If having more type hints means we can rely on the code more, there should be a number associated with that. I propose the term "type coverage" to signify how many variables and other expressions result in `mixed` and which have a more specific type.

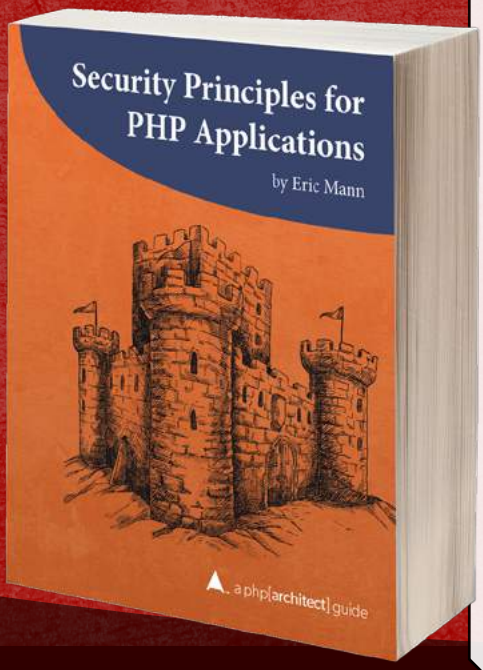
Closing Words

Static analyser should be in the tool belt of every PHP programmer. Similarly to a package manager and unit testing framework, it's an indispensable tool for making quality applications. It should run besides tests on a continuous integration server because its function is similar - to prevent bugs. Once we get used to the feedback from the type system, we can concentrate our testing efforts in places where the static analyzer can't know how the right code should look. In the rest of the application, it has us covered.

Ondřej works as the CTO at Slevomat.cz, moving between product and software development. He enjoys quality assurance and frequent deployment of innovations for the end users. He shares his experience at conferences across Europe, offers his expertise as a consultant, and organizes trainings. He also created PHPStan, a popular static analyzer that focuses on finding bugs in code without running it. @OndrejMirtes

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.

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

Read a Sample Online

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.

php[**architect**]

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

http://phpa.me/mag_subscribe