


 **BACKEND DEVELOPERS WITH SKILLS IN** LARAVEL NODEJS

 **FRONTEND DEVELOPERS WITH SKILLS IN** VUEJS SASS

 **SMART PHONE DEVELOPERS WITH SKILLS IN** REACT NATIVE

 **DEVOPS ENGINEER WITH SKILLS IN** CI&CD LINUX DOCKER



 Follow



Trong bài viết này chúng ta cùng tìm hiểu về PHP Exceptions. Những khái niệm được sử dụng trong nhiều ứng dụng lớn, có thể mở rộng, trong các ứng dụng hướng đối tượng và các frameworks thì việc bắt được các Exception trong quá trình develop là điều cực kì tốt.

Ví dụ đơn giản về exception

Ví dụ sau ta có một function tính diện tích hình tròn, được biểu diễn bằng công thức

$$S = \pi r^2$$

Biểu diễn công thức bằng function sau:

```
function circle_area($radius) {  
    return pi() * $radius * $radius;  
}
```

Trông nó khá đơn giản, tuy nhiên nó không kiểm tra nếu bán kính là một số hợp lệ thì hàm của chúng ta có thể sinh ra lỗi. Bây giờ chúng ta sẽ làm điều đó, và ném một ngoại lệ nếu bán kính là một số âm bằng function như sau:

```
function circle_area($radius) {  
    // bán kính không thể là một số âm  
    if ($radius < 0) {  
        throw new Exception('Invalid Radius: ' . $radius);  
    } else {  
        return pi() * $radius * $radius;  
    }  
}
```

Khi chúng ta gọi hàm với một số âm thì sẽ bắn ra lỗi như sau:



```
1 <?php
2 function circle_area($radius) {
3
4     // bán kính không thể là một số âm
5     if ($radius < 0) {
6         throw new Exception('Invalid Radius: ' . $radius);
7     } else {
8         return pi() * $radius * $radius;
9     }
10 }
11 }
12
13 $radius = -2;
14
15 echo "Circle Radius: $radius => Circle Area: ".
16     circle_area($radius) . "\n";
17
18
19 echo "Another line"
```

```
Terminal
lucct@lucct ~ $ php test.php
PHP Fatal error:  Uncaught exception 'Exception' with message 'Invalid Radius
: -2' in /home/lucct/test.php:6
Stack trace:
#0 /home/lucct/test.php(16): circle_area(-2)
#1 {main}
  thrown in /home/lucct/test.php on line 6
lucct@lucct ~ $
```

Vì nó là một lỗi nghiêm trọng, không thực thi code xảy ra sau đó. Tuy nhiên bạn có thể không luôn luôn muốn kịch bản của bạn để ngăn chặn bất cứ khi nào một ngoại lệ xảy ra. May mắn thay, bạn có thể 'bắt' họ và xử lý chúng. Test các giá trị bán kính với một mảng giá trị và sẽ có các kết quả như sau:

```
1 <?php
2 function circle_area($radius) {
3
4     // bán kính không thể là một số âm
5     if ($radius < 0) {
6         throw new Exception('Invalid Radius: ' . $radius);
7     } else {
8         return pi() * $radius * $radius;
9     }
10 }
11 }
12
13 $radius_array = array(2,-2,5,-3);
14
15 foreach ($radius_array as $radius) {
16
17     try {
18         echo "Circle Radius: $radius => Circle Area: ".
19             circle_area($radius) . "\n";
20     } catch (Exception $e) {
21         echo 'Caught Exception: ', $e->getMessage(), "\n";
22     }
23 }
24 }
```

```
Terminal
lucct@lucct ~ $ php test.php
Circle Radius: 2 => Circle Area: 12.566370614359
Caught Exception: Invalid Radius: -2
Circle Radius: 5 => Circle Area: 78.539816339745
Caught Exception: Invalid Radius: -3
lucct@lucct ~ $
```

Vậy Exception là gì?

Exception đều được xây dựng trong các ngôn ngữ lập trình hướng đối tượng trong một khoảng thời gian khá dài, đối với PHP thì đến phiên bản PHP 5 mới được thông qua.

Exception được hiểu là một ngoại lệ được 'throw' khi có một sự kiện đặc biệt xảy ra. Nó có thể là lỗi phép chia cho số không hoặc bất kì một sự kiện nào đó không hợp lệ và không thực thi được.

Exception thực sự là đối tượng và bạn có tùy chọn để 'bắt' họ và thực thi mã nhất định. Điều này được thực hiện bằng cách sử dụng 'try-catch' khối:

```
try {

    // một vài câu lệnh
    // nơi có thể bắn ra lỗi

} catch (Exception $e) {

    // đoạn code ở đây chỉ được thực thi
    // nếu có lỗi trong khối try ở trên bắn ra

}
```

Chúng ta có thể gửi kèm theo bất kỳ mã trong khối 'try'. Sau đó khối 'catch' được sử dụng cho việc đánh bắt bất kỳ ngoại lệ có thể đã được ném từ bên trong khối try. Khối catch không bao giờ được thực thi nếu không có trường hợp ngoại lệ. Ngoài ra, một khi một ngoại lệ xảy ra, kịch bản ngay lập tức nhảy đến khối catch, mà không thực hiện bất kỳ đoạn code nào nữa.



Khi một ngoại lệ được ném từ một function hoặc một class method, nó đi đến bất function hoặc class đã gọi nó. Và nó vẫn không ngừng làm điều này cho đến khi nó đạt đến đỉnh của ngăn xếp hoặc bị bắt. Nếu nó đạt đến đỉnh của stack và không bao giờ được gọi là, bạn sẽ nhận được một lỗi nghiêm trọng.

For example, here we have a function that throws an exception. We call that function from a second function. And finally we call the second function from the main code, to demonstrate this bubbling effect:

Ví dụ, ở đây chúng ta có một function mà throw ra exception. Chúng ta gọi đó là function từ một function thứ hai. Và cuối cùng chúng ta gọi là function thứ hai từ code chính, để chứng minh hiệu ứng Bubble này:

```
function bar() {  
    throw new Exception('Message from bar().');  
}  
  
function foo() {  
    bar();  
}  
  
try {  
    foo();  
} catch (Exception $e) {  
    echo 'Caught exception: ', $e->getMessage(), "\n";  
}
```

Vì vậy, khi chúng ta gọi foo(), chúng ta cố gắng để nắm bắt bất kỳ trường hợp ngoại lệ có thể. Mặc dù foo() không ném một, nhưng bar() thì có, nó vẫn còn bong bóng lên và bị kẹt ở đầu trang, vì vậy chúng ta nhận được một output: "Caught exception: Message from bar()."

Tracing Exceptions

Since exceptions do bubble up, they can come from anywhere. To make our job easier, the Exception class has methods that allows us to track down the source of each exception.

Kể từ exceptions làm bong bóng lên, họ có thể đến từ bất cứ nơi nào. Để làm cho công việc của chúng ta dễ dàng hơn, các class exception có các phương thức cho phép chúng ta theo dõi nguồn gốc của mỗi exception. Cùng xem một ví dụ có nhiều file và nhiều class.

Đầu tiên chúng ta có một class User và chúng ta save chúng vào file có tên là user.php:

```

class User {

    public $name;
    public $email;

    public function save() {

        $v = new Validator();

        $v->validate_email($this->email);

        // ... save
        echo "User saved.";

        return true;
    }

}

```

Nó sử dụng một class có tên Validator, mà chúng ta đưa vào validator.php:

```

class Validator {

    public function validate_email($email) {

        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new Exception('Email is invalid');
        }

    }

}

```

Từ main code của chúng ta, chúng ta sẽ tạo ra một đối tượng người dùng mới, đặt tên và email của các giá trị. Một khi chúng ta gọi là phương pháp save(), nó sẽ sử dụng các lớp Validator để kiểm tra định dạng email, mà có thể trở lại là một ngoại lệ:

```

include('user.php');
include('validator.php');

$u = new User();
$u->name = 'foo';
$u->email = '$!%#$$%*';

$u->save();

```

However, we would like to catch the exception, so there is no fatal error message. And this time we are going to look into the detailed information about this exception: VIETNAMESE Tuy nhiên, chúng ta muốn bắt ngoại lệ, vì vậy không có thông báo lỗi bắn ra. Và lần này chúng ta sẽ xem xét các thông tin chi tiết về ngoại lệ này:

```
include('user.php');
include('validator.php');
```

VIBLO

```
$u = new User();
$u->name = 'foo';
$u->email = '$!%$%#*';

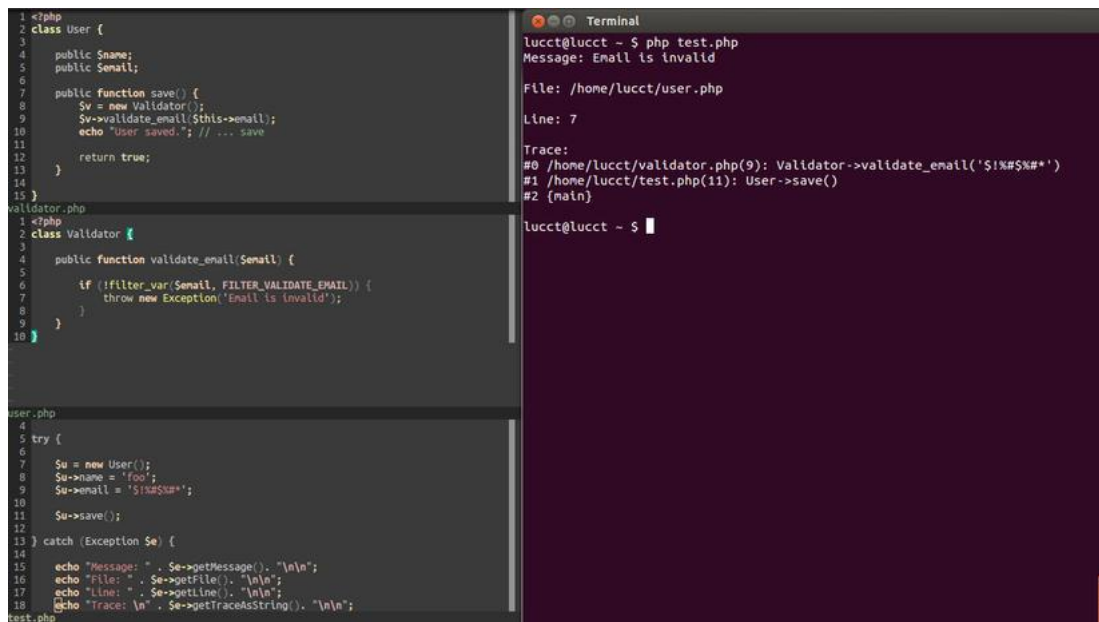
$u->save();

} catch (Exception $e) {

    echo "Message: " . $e->getMessage(). "\n\n";
    echo "File: " . $e->getFile(). "\n\n";
    echo "Line: " . $e->getLine(). "\n\n";
    echo "Trace: \n" . $e->getTraceAsString(). "\n\n";

}
```

Output sẽ hiển thị như sau:



```
1 <?php
2 class User {
3
4     public $name;
5     public $email;
6
7     public function save() {
8         $v = new Validator();
9         $v->validate_email($this->email);
10        echo "User saved."; // ... save
11    }
12    return true;
13 }
14
15 }
16
17 validator.php
18 1 <?php
19 2 class Validator {
20 3
21 4     public function validate_email($email) {
22 5
23 6         if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
24 7             throw new Exception('Email is invalid');
25 8         }
26 9     }
27 10 }
28
29 user.php
30 4
31 5 try {
32 6
33 7     $u = new User();
34 8     $u->name = 'foo';
35 9     $u->email = '$!%$%#*';
36 10
37 11     $u->save();
38 12
39 13 } catch (Exception $e) {
40 14
41 15     echo "Message: " . $e->getMessage(). "\n\n";
42 16     echo "File: " . $e->getFile(). "\n\n";
43 17     echo "Line: " . $e->getLine(). "\n\n";
44 18     echo "Trace: \n" . $e->getTraceAsString(). "\n\n";
45 19 }
46
47 test.php
```

```
Terminal
lucct@lucct ~ $ php test.php
Message: Email is invalid
File: /home/lucct/user.php
Line: 7
Trace:
#0 /home/lucct/validator.php(9): Validator->validate_email('$!%$%#*')
#1 /home/lucct/test.php(11): User->save()
#2 {main}
lucct@lucct ~ $
```

Vì vậy, không cần nhìn vào một dòng code nào, chúng ta có thể cho biết nơi mà các ngoại lệ đến từ đâu. Chúng ta có thể thấy tên file, số dòng, trường hợp ngoại lệ với message và nhiều thông tin hơn nữa. Các dấu vết dữ liệu ngay cả cho thấy dòng chính xác của code đó đã được thực hiện.

Extending Exceptions

Vì đây là một khái niệm hướng đối tượng và ngoại lệ là một lớp, chúng ta thực sự có thể mở rộng nó để tạo ra ngoại lệ tùy chỉnh riêng của chúng ta.

Ví dụ, bạn có thể không muốn hiển thị chi tiết tất cả các ngoại lệ cho người dùng. Thay vào đó, bạn có thể hiển thị một thông điệp thân thiện với người dùng, và đăng nhập các thông báo lỗi nội bộ:

```
// dùng ngoại lệ cho database
class DatabaseException extends Exception {

    // bạn có thể thêm nhiều tùy chỉnh vào trong hàm log
    public function log() {

        // log lỗi tại một vài nơi
        // ...
    }
}

// dùng ngoại lệ cho file
class FileException extends Exception {

    // ...
}
}
```

Chúng ta vừa tạo ra hai loại ngoại lệ mới, trong đó có những tùy chỉnh riêng để phù hợp và thân thiện hơn với người dùng. Khi chúng tôi bắt ngoại lệ, chúng tôi có thể hiển thị một thông điệp cố định, và gọi các phương pháp tùy chỉnh trong nội bộ:

```
function foo() {

    // ...
    // something wrong happened with the database
    throw new DatabaseException();
}

try {

    // put all your code here
    // ...

    foo();

} catch (FileException $e) {

    die ("We seem to be having file system issues.
        We are sorry for the inconvenience.");

} catch (DatabaseException $e) {

    // calling our new method
    $e->log();

    // exit with a message
    die ("We seem to be having database issues.
        We are sorry for the inconvenience.");

} catch (Exception $e) {

    echo 'Caught exception: '. $e->getMessage(). "\n";

}
```

Đây là lần đầu tiên chúng ta nhìn vào một ví dụ với nhiều khối catch cho một khối try duy nhất. Đây là cách bạn có thể bắt các loại khác nhau của các trường hợp ngoại lệ, vì vậy bạn có thể xử lý khác nhau làm cho quá trình phát triển code của chúng ta được sáng sủa hơn.

Bài viết này đơn thuần là giới thiệu về exception trong PHP thuần, tuy nhiên ngày nay công nghệ ngày một phát triển, các framework cũng dần hiện đại hóa, việc bắt các exception thì hầu hết họ đã làm tương đối ổn. Tuy nhiên đối với nhiều dự án chúng ta cần phải hiểu sâu về các kiến thức cơ bản và tự mình có thể customize exception và hỗ trợ việc debug được dễ dàng hơn. Ngoài ra thì việc viết code trong một team thì code của bạn có thể người khác sẽ dùng lại nên code càng sáng sủa bao nhiêu thì thuận tiện cho người sau dùng lại càng dễ dàng hơn.

Tài liệu tham khảo

[Blog.php exception](#)

[PHP manual exception](#)

f

GitHub

9

Atom

