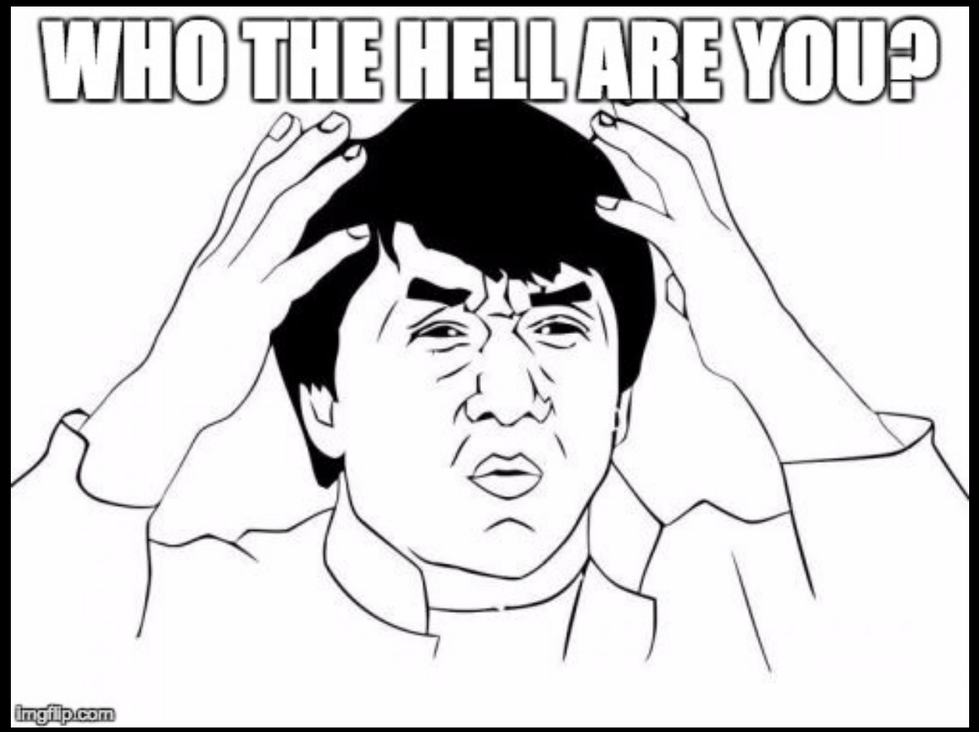
GENERATORS

not just for keeping the lights on


WHO THE HELL ARE YOU?
imgflip.com

DAVID STANLEY

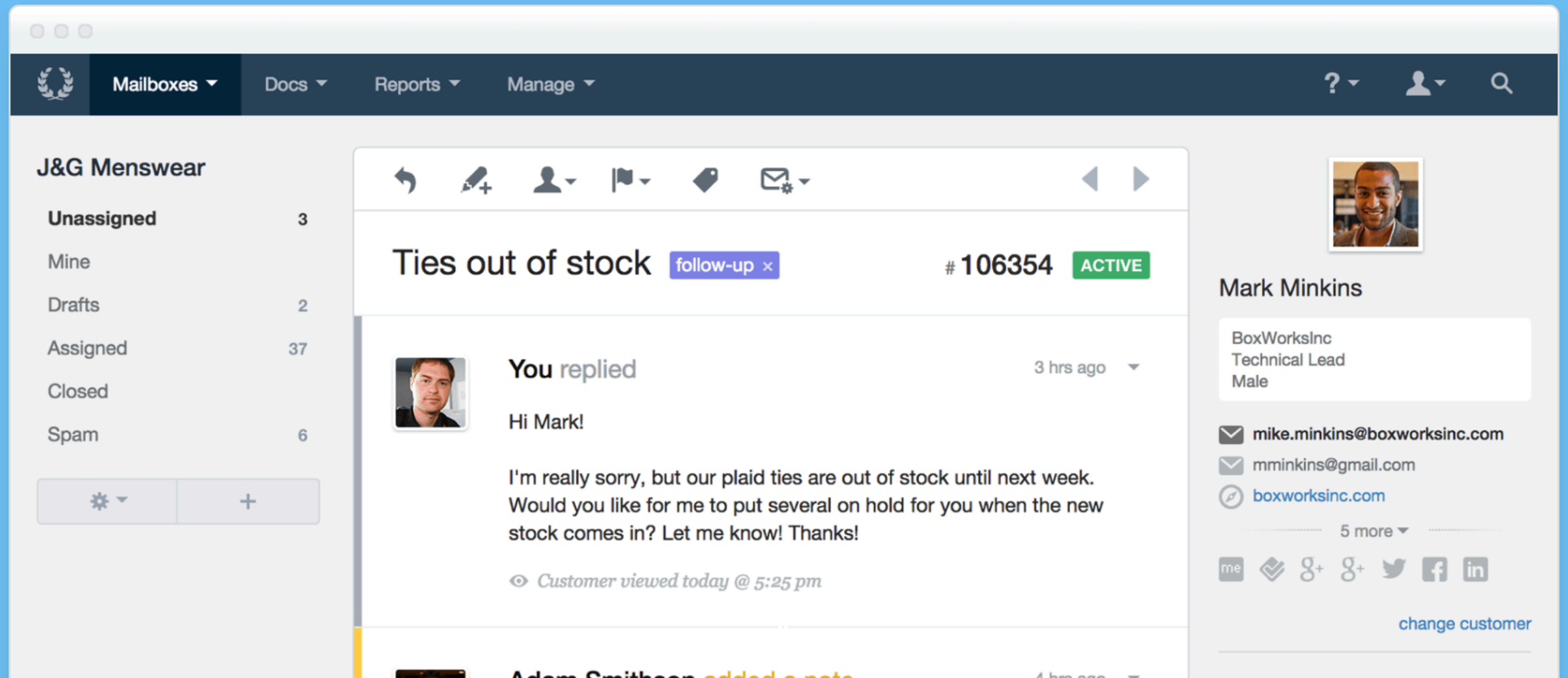
@davidstanley01

[@todo - something personal]

Platform Engineer at Help Scout

A **more human** help desk

Ticket numbers, customer portals & robo-emails aren't a great way to build trust. Help Scout makes every customer interaction a *personalized* one.



DISCLAIMER

- All of the example code can be written in many different ways.
- In many cases, a generator is not the best way to solve a problem.
- Your mileage may vary.

AGENDA

1. PHP Docs definition of a generator
2. Examples from PHP Docs
3. FizzBuzz example
4. Recursively search directory and list file names
5. Data Provider in unit tests

[GENERATORS]

No, not that kind of generator.

From the docs

Generators provide an easy way to implement *simple iterators* without the *overhead* or *complexity* of implementing a class that implements the *Iterator interface*.

Iterate (via *foreach*, for example) over a data set without having to build the entire data set in memory.

Less memory overhead.

Very useful when you have a potentially *large data set* as you only *load what you need* to return the next iteration.

GREAT.

HOW DO I USE THIS BLACK MAGIC?

With PHP 5.5, we got a new keyword.

A *generator* is a function that *yields* a value rather than returning one.

```
// this
function i_am_a_generator()
    yield range(1, 10);
}

// not this
function i_am_not_a_generator()
    return range(1, 10);
}
```

WTF?

In essence, the *yield* keyword 'pauses' the execution of the generator until the next value is requested.

As a *coroutine*, this means that execution responsibility is passed between the generator and the calling scope.

SIMPLE EXAMPLE...

TAKEN FROM PHP DOCS - GENERATOR SYNTAX

RE-IMPLEMENT `range()` USING A GENERATOR

```php
echo 'Single digit odd numbers from range():  ';
foreach (range(1, 9, 2) as $number) {
    echo "$number ";
}

// Single digit odd numbers from range():  1 3 5 7 9
```

```php
function xrange($start, $limit, $step = 1) {
    if ($start < $limit) {
        if ($step <= 0) {
            throw new LogicException('Step must be +ve');
        }

        for ($i = $start; $i <= $limit; $i += $step) {
            yield $i;
        }
    } else {
        if ($step >= 0) {
            throw new LogicException('Step must be -ve');
        }

        for ($i = $start; $i >= $limit; $i += $step) {
            yield $i;
        }
    }
}

echo 'Single digit odd numbers from xrange(): ';
foreach (xrange(1, 9, 2) as $number) {
    echo "$number ";
}

// Single digit odd numbers from range():  1 3 5 7 9
```

THAT WASN'T EASIER!

You're right.

HOW ABOUT THIS ONE?

```php
function gen_zero_to_three() {
    for ($i = 0; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $i;
    }
}

$generator = gen_zero_to_three();
foreach ($generator as $value) {
    echo "$value";
}

// 0 1 2 3
```

```php
class ContrivedExample implements Iterator
{
    protected $set = [];
    protected $position = 0;

    public function __construct($reps) {
        $this->set = range(, $reps);
    }

    public function rewind()  {  }

    public function valid() {
        return isset($this->set[$this->position]);
    }

    public function current() {
        return $this->set[$this->position];
    }

    public function key() {
        return $this->position;
    }

    public function next() {
        $this->position++;
    }
}

$generator = new ContrivedExample(3);
while ($generator->valid()) {
    echo $generator->current() . " ";
    $generator->next();
}

// 0 1 2 3
```

You can `yield` from a *generator* calling *other generators*!

[insert tired Xzibit *Yo Dawg!* meme]

```php
function count_to_fifteen() {
    yield 1;
    yield 2;
    yield from [3, 4];
    yield from new ArrayIterator([5, 6]);
    yield from seven_eight();
    yield 9;
    yield 10;

    // Any Traversable object can be used!
    yield from (new Haystack\HArray(range(11, 15)));
}

function seven_eight() {
    yield 7;
    yield from eight();
}

function eight() {
    yield 8;
}

foreach (count_to_fifteen() as $num) {
    echo "$num ";
}

// 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

yield vs return

Processing *stops* when `return` is called,
and the local scope is *destroyed*.

```php
function fibonacci($item) {
    $a = 0;
    $b = 1;
    for ($i = 0; $i < $item; $i++) {
        return $a;

        // never gets called!
        $a = $b - $a;
        $b = $a + $b;
    }
}

$fibo = fibonacci(10);
foreach ($fibo as $value) {
    echo "$value ";
}

// WARNING Invalid argument supplied for foreach() on line number 15
// That's cuz the function returned an integer and not an iterable entity

var_dump($fibo);
// int(0)
```

Processing *pauses* when `yield` is called,
but the local scope is *preserved*.

```php
function fibonacci($item) {
    $a = 0;
    $b = 1;
    for ($i = 0; $i < $item; $i++) {
        // execution pauses here and returns $a
        yield $a;

        // When control is passed back, execution will
        // resume here and proceed to the next yield
        // statement
        $a = $b - $a;
        $b = $a + $b;

        // tricky!!
        if ($i + 1 !== $item) {
            yield ' - ';
        }
    }
}

$fibo = fibonacci(10);
foreach ($fibo as $value) {
    echo "$value ";
}
// 0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34

var_dump($fibo);
// object(Generator)#1 (0) { }

// http://php.net/manual/en/language.generators.syntax.php#117460
```

WHAT THE HELL IS THAT??

GENERATOR CLASS

Internal class that implements the `Iterator` interface, but is *forward-only*.

Can't rewind it.
Can't create a new object via `new` keyword.

```php
function simple() {
    yield from range(0, 10, 2);
}

$gen = simple();

var_dump($gen);
// object(Generator)#1 (0) { }
```

```
Generator implements Iterator {
    /* Methods */
    public mixed current ( void )
    public mixed getReturn ( void )
    public mixed key ( void )
    public void next ( void )
    public void rewind ( void )
    public mixed send ( mixed $value )
    public mixed throw ( Exception $exception )
    public bool valid ( void )
    public void __wakeup ( void )
}
// http://php.net/manual/en/class.generator.php
```

```php
function simple() {
    yield from range(0, 10, 2);
}

// This...
$gen = simple();
foreach ($gen as $val) {
    echo "$val ";
}
// 0 2 4 6 8 10

// ...is the same as this.
$nextGen = simple();
while ($nextGen->valid()) {
    echo $nextGen->current() . " ";
    $nextGen->next();
}
// 0 2 4 6 8 10
```

What is that `send` method?

The `send` method lets you inject a value into the generator and then resume execution.

```php
function printer() {
    while (true) {
        $string = yield;
        echo $string;
    }
}

$printer = printer();
$printer->send('Hello world!');
echo " - ";
$printer->send('Bye world!');

// Hello world! - Bye world!
```

DOWN THE RABBIT HOLE...

The `send` method opens the door to things like cooperative multitasking, scheduling, coroutines and interesting socket communications.

Read this post from Nikita Popov for more information.


DO IT LIVE!

FIZZBUZZ

Write a program that prints the numbers from 1 to 100. But, for multiples of three, print “Fizz” instead of the number. For the multiples of five, print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

-- angry hiring manager

RECURSION WITH A GENERATOR

Given a directory, search through it and list all files contained within (as well as those file contained with subdirectories).

Example courtesy of @magnetikonline

DATAPROVIDERS

This is more of a party trick than anything else...

GIVEN THESE ENTITIES...

```php
class Book
{
    /** @var string GUID */
    private $id;

    /** @var string */
    private $name;

    /** @var string */
    private $isbn;

    /** @var Author */
    private $author;

    /** @var boolean */
    private $fiction = false;

    // ... imagine the getters and setters...
}

class Author
{
    /** @var string */
    private $name;

    /** @var Publisher */
    private $publisher;

    /** @var Book[] */
    private $books = [];

    // ... imagine the getters and setters...
}

class Publisher
{
    /** @var string */
    private $name;

    /** @var Author[] */
    private $authors = [];

    // ... imagine the getters and setters...
}
```

AND GIVEN THIS MESS...

```php
class BookHelper
{
    public function getBooksForPublisher(Publisher $publisher)
    {
        $authors = $publisher->getAuthors();

        $authorBooks = array_map(function(Author $author) {
            return $author->getBooks();
        }, $authors);

        $books = array_map(function(Book $book) {
                return [
                    $book->getId() => [
                        'isbn'   => $book->getIsbn();
                        'author' => $book->getAuthor()->getName(),
                        'title'  => $book->getName(),
                        'genre'  => $book->isFiction() ? 'fiction' : 'non-fiction'
                    ]
                ];
            }, $authorBooks);

        return call_user_func_array('array_merge', $books);
    }
}
```

WRITE SOME TESTS TO COVER *all* OF THE EXECUTION PATHS

IGNORE THE OBVIOUS DESIGN FLAWS

TYPICAL TEST MIGHT LOOK LIKE THIS...

```php
class BookHelperTest extends PHPUnit_Framework_TestCase
{
    public function bookHelperProvider()
    {
        return [
            [
                'isbn'       => 'this-is-an-isbn-number',
                'author'     => 'J.K. Rowlings',
                'id'         => 'not-really-a-guid',
                'title'      => 'Harry Potter and the Cash Grab',
                'isFiction'  => true
            ],
            ['isbn' => '', 'author' => '', 'id' => '', 'title' => '', 'isFiction' => '']
        ];
    }

    /** @dataProvider bookHelperProvider */
    public function testBookHelperReturnsArray($isbn, $author, $id, $title, $isFiction)
    {
        $expected = [
            $id => [ 'isbn' => $isbn, 'author' => $author, 'title'  => $title, 'genre' => 'fiction' ],
        ];

        $author = new Author();
        $author->setName($author);

        $book = new Book();
        $book->setId($id);
            ->setTitle($title)
            ->setIsbn($isbn)
            ->setAuthor($author)
            ->setIsFiction($isFiction);
        $author->addBook($book);

        $publisher = new Publisher();
        $publisher->addAuthor($author);

        $helper = new BookHelper();
        $result = $helper->getBooksForPublisher($publisher);

        $this->assertEquals($expected, $result);
    }
}
```

TOO MUCH FOR ME TO READ

```php
class BookHelperTest extends PHPUnit_Framework_TestCase
{
    public function bookHelperProvider()
    {
        return [
            [
                'isbn'       => 'this-is-an-isbn-number',
                'author'     => 'J.K. Rowlings',
                'id'         => 'not-really-a-guid',
                'title'      => 'Harry Potter and the Cash Grab',
                'isFiction'  => true
            ],
            ['isbn' => '', 'author' => '', 'id' => '', 'title' => '', 'isFiction' => '']
        ];
    }

    /** @dataProvider bookHelperProvider */
    public function testBookHelperReturnsArray($isbn, $author, $id, $title, $isFiction)
    {
        $expected = [
            $id => [ 'isbn' => $isbn, 'author' => $author, 'title'  => $title, 'genre' => 'fiction' ],
        ];

        $author = new Author();
        $author->setName($author);

        $book = new Book();
        $book->setId($id);
            ->setTitle($title)
            ->setIsbn($isbn)
            ->setAuthor($author)
            ->setIsFiction($isFiction);
        $author->addBook($book);

        $publisher = new Publisher();
        $publisher->addAuthor($author);

        $helper = new BookHelper();
        $result = $helper->getBooksForPublisher($publisher);

        $this->assertEquals($expected, $result);
    }
}
```

USING A GENERATOR AS A DATA PROVIDER...

```php
class BookHelperTest extends PHPUnit_Framework_TestCase
{
    public function getHelperData()
    {
        yield from [
            [
                'isbn'      => 'this-is-an-isbn-number',
                'author'    => 'J.K. Rowlings',
                'id'        => 'not-really-a-guid',
                'title'     => 'Harry Potter and the Cash Grab',
                'isFiction' => true,
                'genre'     => 'fiction'
            ],
            ['isbn' => '', 'author' => '', 'id' => '', 'title' => '', 'isFiction' => '', 'genre' => '']
        ];
    }

    public function bookHelperProvider()
    {
        foreach ($this->getHelperData() as $row) {
            extract($row); // $isbn, $author, $id, $title, $isFiction, $genre

            $author = new Author();
            $author->setName($author);

            $book = new Book();
            $book->setIsbn($isbn)
                ->setTitle($title)
                ->setAuthor($author)
                ->setIsFiction($isFiction);

            $publisher = new Publisher();
            $publisher->addAuthor($author);

            $expected = [
                $id => [ 'isbn' => $isbn, 'author' => $author, 'title'  => $title, 'genre' => $genre ],
            ];

            yield [$publisher,  $expected];
        }
    }

    /** @dataProvider bookHelperProvider */
    public function testBookHelperReturnsArray(Publisher $publisher, $expected)
    {
        $helper = new BookHelper();
        $result = $helper->getBooksForPublisher($publisher);
        $this->assertEquals($expected, $result);
    }
}
```

SUMMARY

1. Generators trade *speed* for *memory* footprint
2. Generators become *more valuable* as the size of the *data set grows*
3. Generators are syntactic sugar around a *simple iterator*

```
yield from $questions
```