www.phparch.com

October 2014
VOLUME 13 - ISSUE 10

Built With PHP

php[architect]

FREE SAMPLE ARTICLE

ALSO INSIDE

Editorial:
Building Blocks

Community Corner:
October 2014

Laravel:
Deploying Applications
Part 2: Automation

finally{}:
Types of Open-Source
Software Projects

We Built DataSift on PHP

Protect Your Data with ownCloud

Domain Modeling with PHP in Polyglot Systems

Test Fixtures Like a Boss

Varnish: Just Plain Faster

WordPress, Drupal, Joomla!, Magento, Zend Framework, Symfony, Laravel, more...

PHP[WORLD]

November 10-14th 2014

Washington, D.C.

Five Amazing Keynotes...

Luke Stokes
Co-founder and CTO of FoxyCart.com

Angela Byron
Drupal Core Committer

Andrew Nacin
WordPress Lead Developer

Jeffrey McQuire
Open Source Evangelist at Acquia

The Greatest Panel on Earth!

We will have key people representing seven of the biggest PHP frameworks and applications: WordPress, Drupal, Magento, Joomla!, Symfony, Laravel, and Zend Framework all together in one place to answer all your most difficult questions!

Find out more and buy your ticket at world.phparch.com

AUTOMATTIC MANDRILL CODE CLIMATE

in2it professional php services

rackspace the #1 managed cloud company

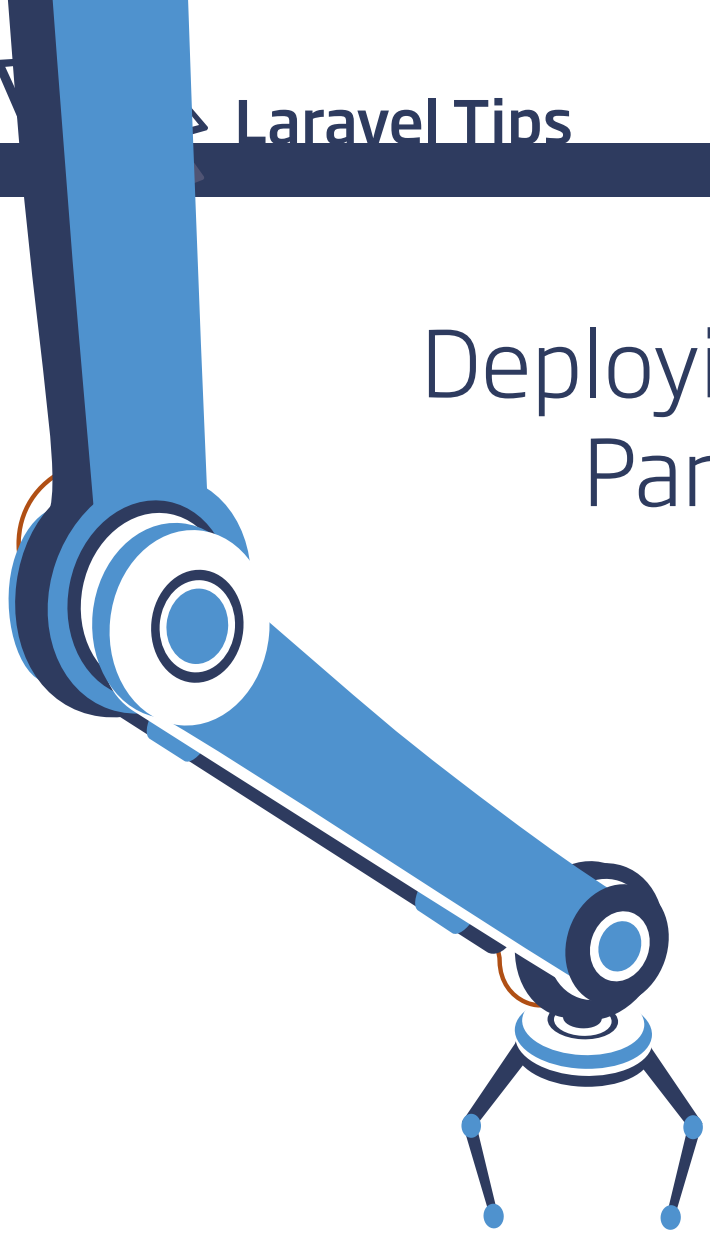
MySQL

REDPORT INFORMATION ASSURANCE

BLACKMESH TURN 14 DISTRIBUTION Engine Yard ACQUIA

Deploying Applications Part 2: Automation

Dirk Merkel



Whether you are developing in a local virtual machine with Laravel's Homestead or deploying code to AWS, automation is the name of the game. Exploring Laravel's support for SSH, we will learn how to automate the updating and deploying of applications to save time and minimize downtime.

DisplayInfo()

Requirements:

- PHP: 5.3.7+
- MCrypt
- Composer
- Laravel 4.2

Related URLs:

- Laravel PHP Framework - http://laravel.com
- GitHub - https://github.com
- Invoice Ninja - https://www.invoiceninja.com
- Invoice Ninja source code on GitHub - https://github.com/hillelcoren/invoice-ninja
- VirtualBox - https://www.virtualbox.org
- Vagrant - http://www.vagrantup.com
- Laravel Homestead Documentation - http://laravel.com/docs/homestead
- Laravel Homestead Code on GitHub - https://github.com/laravel/homestead.git
- Amazon Web Services - https://aws.amazon.com
- Laravel Documentation: Artisan Development - http://laravel.com/docs/commands
- Laravel Documentation: SSH - http://laravel.com/docs/ssh

Introduction

Having to deploy your code to development, test, QA, staging, CI, and production environments can be a chore. What's worse is that you're more prone to make a mistake if you have to do so manually. However, you don't have to add a devops team member because Laravel has a solution that we can use for just this purpose. By bringing up an EC2 instance in AWS and deploying to it, we will explore Laravel's support for SSH, which can be used to automate various tasks, application deployment included. Wrapping SSH tasks into Artisan commands allows us to execute these tasks easily from the command line.

Recap

In part 1 of this series (see the September 2014 issue), we set up a local development environment for the open-source Invoice Ninja project. We did this using Laravel's officially supported Vagrant box, Homestead. This approach allows us to separate our development environment and all of its dependencies, such as specific versions of PHP, Apache HTTP, MySQL, etc., from our main OS, while still keeping everything local. The shared source code directory allows us to use our preferred editor installed on our own machine to write the code while having PHP and Apache serve the site and execute the same code from within the VM. In my case, I'm using Sublime Text to edit the Invoice Ninja project. I then use Google Chrome installed on my MacBook to view the site as it is being served by PHP 5.5.x and Nginx on the Ubuntu 14.04-based virtual machine of Homestead.

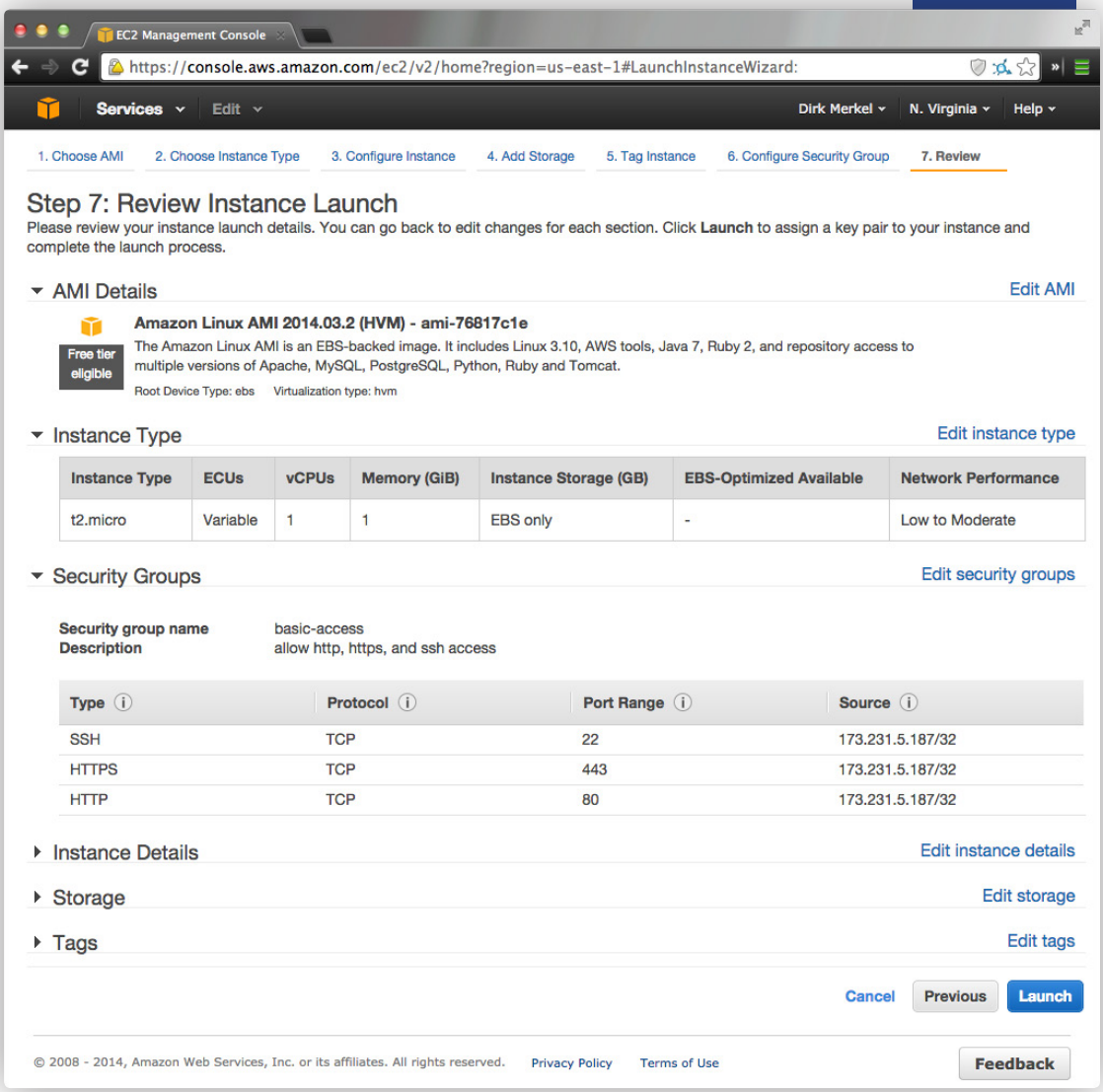
To the Cloud

Let's add a production environment to which we can deploy the application, preferably with the help of some automation. I've been making pretty heavy use of Amazon Web Services (AWS) for several years now and that is where we will be hosting the production instance of the application. Luckily, AWS's console provides a step-by-step wizard for launching a virtual machine. Figure 1 shows the summary screen just before I clicked the *Launch* button. There are three notable choices I have made for the new EC2 instance. First, I picked *Amazon Linux AMI 2014.03.2 (HVM)* to be the OS on my VM. This is AWS's in-house distribution of Linux so it guarantees high compatibility with AWS services and infrastructure, along with the lowest cost. Ubuntu 14.0.4 would have been another good choice as it is the same OS that underlies Homestead. Other options, including Red Hat Enterprise Linux and Microsoft Windows Server 2012, are less ideal and come with a licensing cost added to the hourly cost of the VM.

Second, I picked a *t2.micro* instance. The t2 family of AMIs was introduced this year and represents a low-cost choice for non-constant and burstable performance. The *t2.micro* instance I chose for this example runs at $0.013 per hour or $113.88 per year. Additional savings are possible when committing to Reserved Instances as opposed to Spot Instances.

Third, I created a Security Group, which I called *basic-access*. It limits access to the server to ports 22 (SSH), 80 (HTTP), and 443 (HTTPS). This will allow access to the hosted site(s) with a browser on ports 80 and 443, as well as access to the server's terminal via SSH on port 22. You can add additional ports, such as port 3306 for MySQL, but keep in mind that this server is on the Internet and you should keep your attack surface as minimal as possible.

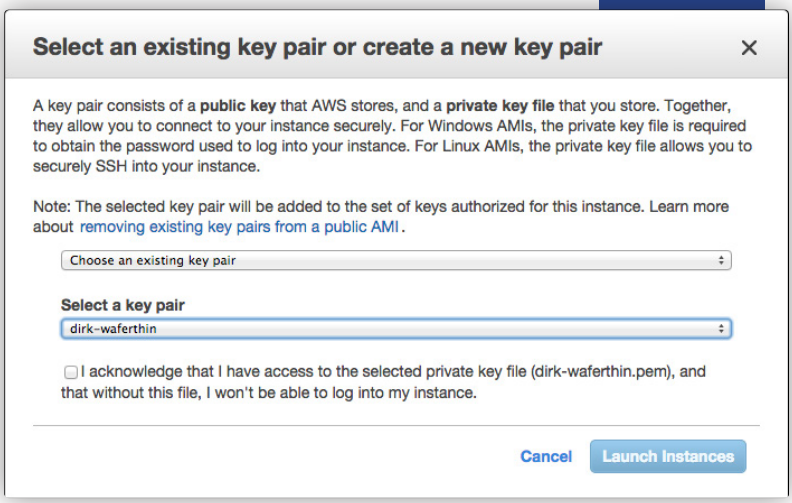
AWS EC2 Instance Launch Wizard **FIGURE 1**



A summary of the EC2 instance's configuration can be seen in Figure 1.

Lastly and most importantly for our purposes, I selected an existing key pair. We are talking about a public-private key pair where the public part gets placed on the server and I keep the private one. This will allow us to connect to the EC2 instance via SSH. You can see in Figure 2 that I selected an existing key pair that I had generated previously. If you don't yet have a key pair, the AWS interface will let you generate and download one on-the-fly.

AWS EC2 Key Pair Selection **FIGURE 2**



After clicking the *Launch* button, AWS takes about a minute to boot up the instance. At this point we can SSH into the machine using the key pair we selected in the previous step. To continue our example, I quickly went through the installation steps for the Invoice Ninja site, which I can now access from the browser using the instance's public IP address. You can refer to part one of this article or the Invoice Ninja site for step-by-step installation instructions.

Let's Automate

With our cloud instance set up, let's edit our local project so we can communicate with it and automate some remote command execution. Invoice Ninja, being a Laravel project, has a file `app/config/remote.php`, where we can specify any remote servers with which we intend to interact. Putting the connection info for the instance we just launched in AWS, we get the file shown in Listing 1.

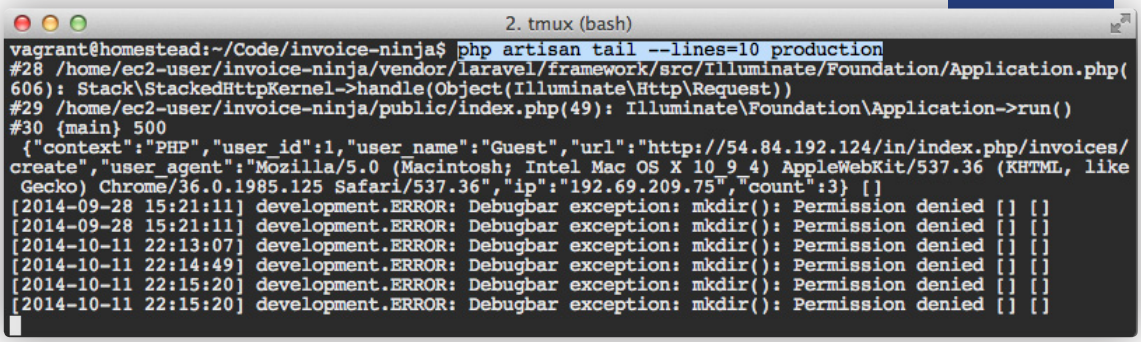
The host IP address of `54.84.192.124` is the public IP of the EC2 instance. The default user name for all EC2 instances is `ec2-user`, which we assign to the `username` keyword. It also shows up in the `root` keyword, which specifies the starting directory on the server. The `dirk-waferthin.pem` file contains the private portion of the key pair, but like other key pairs generated by AWS, it has no `keyphrase`.

Also note that we named this configuration `production` and listed the production server in the `web` group at the end of Listing 1. Groups allow you to execute the same command on all servers in a group. This is obviously useful when you want to update all the servers in your production environment at the same time, for example.

LISTING 1

```
<?php
return array(
    /*
    |--------------------------------------------------------------------------
    | Default Remote Connection Name
    |--------------------------------------------------------------------------
    |
    | Here you may specify the default connection that will
    | be used for SSH operations. This name should correspond
    | to a connection name below in the server list. Each
    | connection will be manually accessible.
    */
    'default' => 'production',
    /*
    |--------------------------------------------------------------------------
    | Remote Server Connections
    |--------------------------------------------------------------------------
    |
    | These are the servers that will be accessible via the
    | SSH task runner facilities of Laravel. This feature
    | radically simplifies executing tasks on your servers,
    | such as deploying out these applications.
    |
    */
    'connections' => array(
        'production' => array(
            'host'      => '54.84.192.124',
            'username'  => 'ec2-user',
            'password'  => '',
            'key'       => '/home/vagrant/Code/invoice-ninja'
                         . '/app/config/dirk-waferthin.pem',
            'keyphrase' => '',
            'root'      => '/home/ec2-user',
        ),
    ),
    /*
    |--------------------------------------------------------------------------
    | Remote Server Groups
    |--------------------------------------------------------------------------
    |
    | Here you may list connections under a single group
    | name, which allows you to easily access all of the
    | servers at once using a short name that is extremely
    | easy to remember, such as "web" or "database".
    |
    */
    'groups' => array(
        'web' => array('production')
    ),
);
```

Remote Tail With Artisan **FIGURE 3**



Now that we have defined our remote configuration, we can actually use it without having to do anything else. Artisan, Laravel's trusty sidekick and command-line interface, has the built-in `tail` command, which we can use to remotely tail a log file. Figure 3 shows the output from the command using the `remote.php` configuration from Listing 1.

Being able to tail a remote log file is nice, but let's take the next step by actually running some commands on the remote server. We'll create an Artisan command to update the remote server with the latest revision from GitHub and run any database migrations. This command can then be used to quickly update one or more remote servers, for example to deploy the most recent release to the production environment.

We start by having Artisan create a new command. Running the following Artisan command will create an empty class, which will serve as a template for creating our own Artisan command (all on one line):

```
php artisan command:make UpdateInstance --command=remote:update
```

LISTING 2

```
01. <?php
02. use Illuminate\Console\Command;
03. use Symfony\Component\Console\Input\InputOption;
04. use Symfony\Component\Console\Input\InputArgument;
05.
06. class UpdateInstance extends Command {
07.     /**
08.      * The console command name.
09.      * @var string
10.      */
11.     protected $name = 'remote:update';
12.     /**
13.      * The console command description.
14.      * @var string
15.      */
16.     protected $description = 'Given a remote connection as
17.       argument, this command will connect to the
18.       corresponding server, update the installed version of
19.       the application to the latest revision in GitHub\'s
20.       master branch, and run any migrations.';
21.
22.     /**
23.      * Create a new command instance.
24.      * @return void
25.      */
26.     public function __construct() {
27.         parent::__construct();
28.     }
29.
30.     /**
31.      * Execute the console command.
32.      * @return mixed
33.      */
```

Continued Next Page

LISTING 2 (CONT'D)

```
34.    public function fire() {
35.       // connection against which to execute command
36.       $connection = $this->argument('connection');
37.        // commands to execute
38.       $commands = array(
39.                         'cd /home/ec2-user/invoice-ninja',
40.                         'git pull origin',
41.                         'php artisan cache:clear',
42.                   );
43.        // whether to run the migrations
44.       if ($this->option('migrate')) {
45.            $commands[] = 'php artisan migrate';
46.       }
47.        // execute the commands against the connection
48.       SSH::into($connection)->run(
49.          $commands,
50.          function($line) {
51.             // display output from remote command if
52.             // verbose option is given
53.             if ($this->option('verbose')) {
54.                // output each line
55.                $this->info($line);
56.             }
57.          }
58.       );
59.    }
60.
61.     /**
62.      * Get the console command arguments.
63.      * @return array
64.      */
65.    protected function getArguments() {
66.       return array(
67.          array('connection',
68.                InputArgument::OPTIONAL,
69.                'Environment to update.',
70.                'production'
71.          ),
72.       );
73.    }
74.     /**
75.      * Get the console command options.
76.      * @return array
77.      */
78.    protected function getOptions() {
79.       return array(
80.          array('migrate',
81.                null,
82.                InputOption::VALUE_NONE,
83.                'Run the migrations via Artisan.',
84.                null
85.          ),
86.       );
87.    }
88. }
```

By default, Artisan creates the `UpdateInstance` class in the `app/command/` directory, but that default can be overwritten with the `--path` option. The `UpdateInstance` class extends Laravel's `Command` class, which in turn derives from Symfony's powerful console components. Because the parent classes do much of the heavy lifting for us, we only have to implement a couple of methods to create our full-fledged Artisan command.

Listing 2 shows the final `UpdateInstance` class.

The `$name` and `$description` properties contain the name with which to invoke the command and a short description, respectively. After registering the command with Artisan, we will see these two properties appear in Artisan's list of available commands. The `getArguments()` and `getOptions()` methods let us specify command arguments and options that Artisan will automatically parse from the command line and make available to our code via `argument()` and `option()` accessor methods. It will even generate error messages for missing required arguments.

Both arguments and options are defined in their respective arrays. Options use the following values:

```
array($name, $mode, $description, $defaultValue)
```

And arguments use the following values:

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

We have one optional argument, `connection`, which defaults to `production` if not given. `connection` indicates the server against which to execute the commands. Our only option is `migrate`, which defaults to `null`, and indicates whether to run the database migrations after the code update.

Now we get to the core of the remotely executing code. The `fire()` method contains the code that will get executed when we call the `remote:update` command via Artisan. First, we get the `$connection` information. Second, we create an array of the commands to execute on the remote server. If the `migrate` option was given on the command line, we add the `migrate` Artisan command to our array.

Remote execution of commands is provided via Laravel's SSH facade. The `into()` method lets us specify the connection against which we then `run()` the list of commands. `run()` optionally accepts a Closure as the second argument, which we use to print the remote command output to the local console if the `verbose` option was given. Note that we didn't have to define the `verbose` option since it comes with all generated Artisan commands.

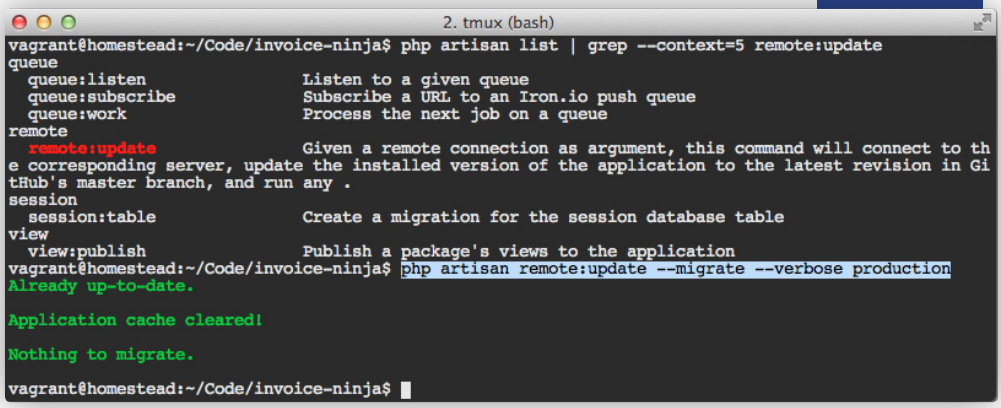
Before actually running the command, we need to let Artisan know about it, which we do by adding the following line to the `app/start/artisan.php` file:

```
Artisan::add(new UpdateInstance);
```

The results of executing the command via Artisan can be seen in Figure 4. First we ask Artisan to list all known commands and grep for the `remote:update` command we just created. Seeing the command listed, we proceed to execute the command via Artisan against the production environment.

From the output to the screen, we can see that the commands were executed. However, there had been no code commits to GitHub and consequently no database migrations needed to be run.

Remote Command Execution With Artisan **FIGURE 4**



Conclusion

In this installment of our deployment automation series of articles, we created a cloud environment for our application in AWS. We then configured a connection in Laravel that allowed us to access that environment programmatically. After using a built-in Artisan command to tail a remote log file on our cloud-based instance, we created our own Artisan command that used Laravel's SSH facade to execute a series of commands on the remote server to update the code, run database migrations, and flush the application cache.

These examples should provide a great starting point for you to explore the topic further by creating your own automation tasks and commands. Consider the possibilities of grouping connections and scripting more complex tasks.

In the next installment in this series, we will look at moving our remote automation tasks to Envoy. While Laravel's SSH facade is merely a wrapper around the underlying SSH protocol, Envoy is a full-fledged remote task runner. We will explore some of the benefits of Envoy while continuing to work through automation examples in our cloud-based environment.

DIRK MERKEL is the CTO for Vivantech Inc. and has experience architecting solutions and managing the software development process in large and small organizations. His focus is on Open Source and often web-centric technologies, including Java, PHP, Perl, Ruby, MySQL, Apache, etc.

g+ http://bit.ly/phpa_DirkMerkel

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine

books

conferences

training

phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.