www.phparch.com

August 2016
VOLUME 15 - Issue 8

php[architect]

Unleashing Automation

Deploying with Ansible

Using Etsy/Phan for Static Analysis

How We Automate With Slack

ALSO INSIDE

More Than Just an "OK" Dev

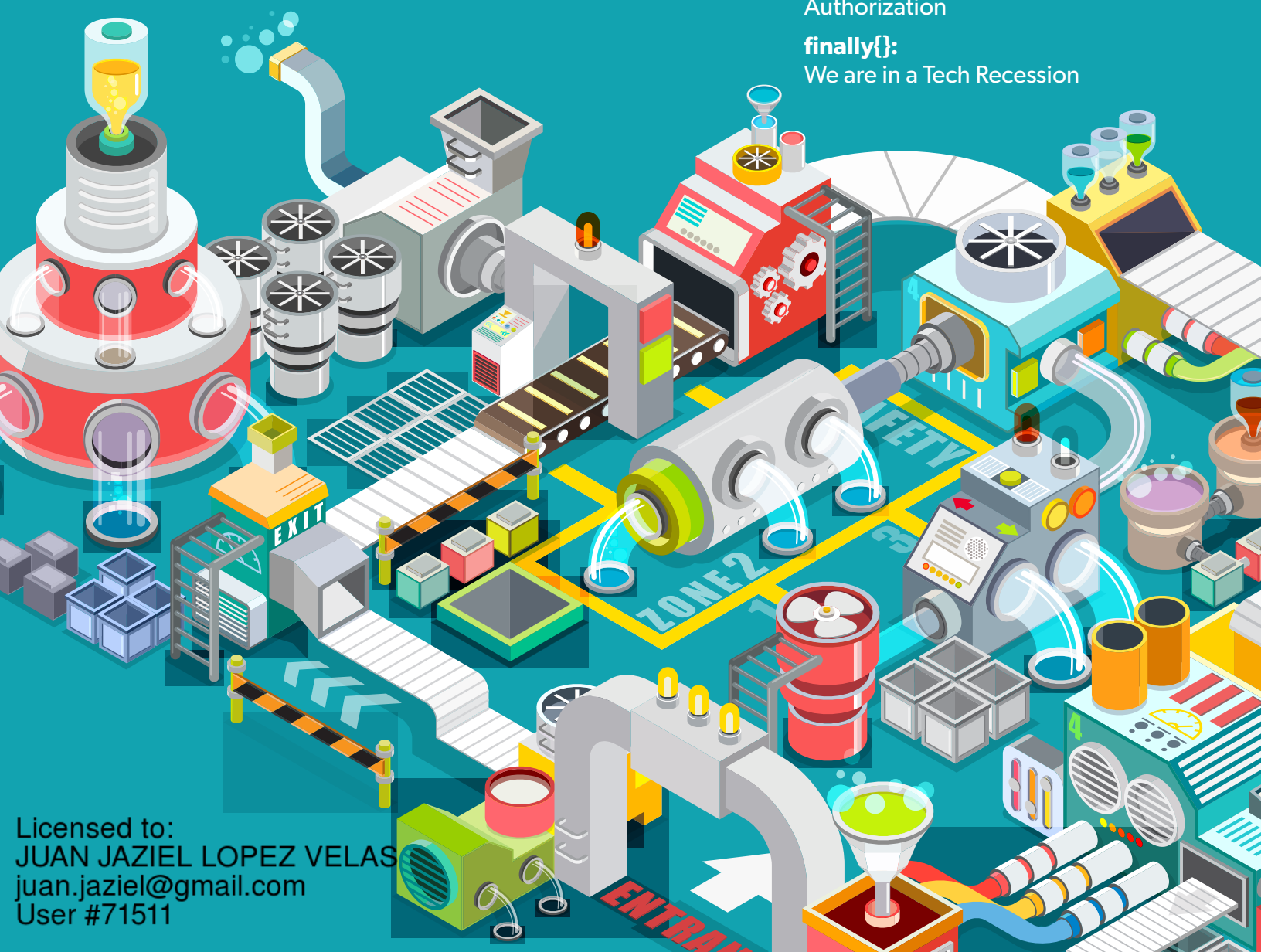
What's Your Code Tolerance?

Connecting Your Charts to a MySQL Database

Community Corner:
The Tangled Web We Weave

Security Corner:
The Hitchhiker's Guide to Authorization

finally{}:
We are in a Tech Recession

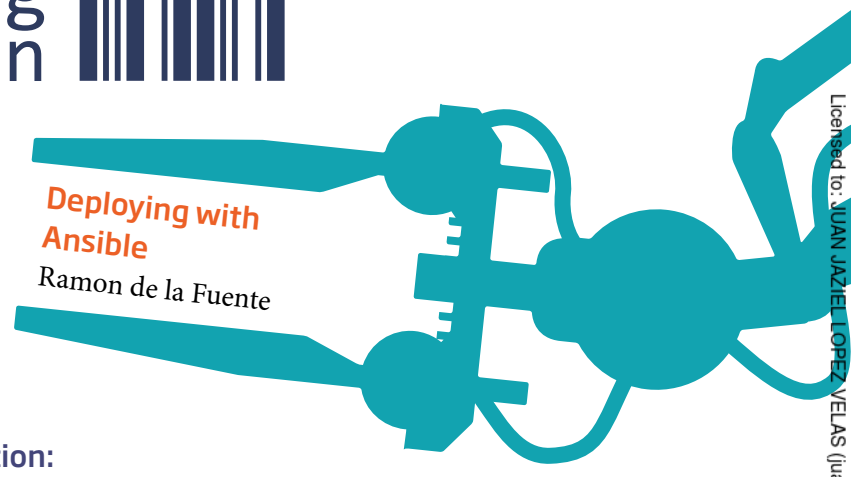


php[architect]

CONTENTS

AUGUST 2016
Volume 15 - Issue 8

Unleashing Automation



Columns

Editor-in-Chief: Oscar Merida

Creative Director: Kevin Bruce

Technical Editors:
Oscar Merida, Sandy Smith

Issue Authors:
Steve Bennett, Chris Cornutt, Ramon de la Fuente, Cal Evans, Vikas Lalwani, Gabriel Somoza, David Stockton, Matthew Setter, Eli White

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

Unleashing Automation

One of my unstated resolutions for this year has been to automate repetitive tasks as much as possible. There is always more to do than time available, and one way to free up time is to have a computer do it for us. Not only will the computer do them much faster, but they won't make mistakes if you write them correctly. Since then, I've been using automated tests with Behat, building command line tools with Symfony, and setting up LiveTemplates in PhpStorm for bits of text or code that I use frequently. I still have a lot on my plate, but it's helped me focus on important tasks instead of losing time on tedious ones.

This month, we have articles to help you manage ever-more-complex systems with tools to help you automate any repetitive and error prone tasks. First, in *Deploying with Ansible*, Ramon de la Fuente will guide you through setting up a foolproof deployment process for application updates. If you're still using a manual process to update the code and database for your website, he'll share an adaptable approach to deployments and show you how to implement it with Ansible. Chatops has gained popularity because of how easy it makes complicated tasks manageable from the chat interfaces we use. David Stockton shares how he's automated development processes at his job using Slack, Webhooks, and reusable middlewares in *Leveling Up: How We Automate With Slack*. Last, if you're looking to migrate to PHP 7, check out *Education Station* where Matthew Setter looks at *Using Etsy/Phan for Static Analysis*. Specifically, as a tool for automatically scanning your code base to check if it's compatible with PHP 7.

Also in this issue, Gabriel Somoza shares his advice on being *More Than Just an "OK" Dev*. If you want to prepare your resume to stand out when you're looking for your next career move, he has some tips to keep you competitive. In *Connecting Your Charts to a MySQL Database*, Vikas Lalwani provides a prototype for generating dynamic charts. You'll see how to use FusionCharts to render client-side charts with JavaScript and feed it data points from a database with PHP. Looking to make code reviews less stressful for you and your colleagues? Steve Bennet helps you understand *What's Your Code Tolerance?* and how to apply it when reviewing pull requests.

In our regular columns this month, Cal Evans in *Community Corner* asked Samantha Quiñones to share her thoughts about community. She shares how the friends and connections she's made by being active impacted her life on a recent road trip and beyond. Chris Cornutt explores different ways to make sure your users have the right permissions in *Security Corner: The Hitchhiker's Guide to Authorization*. There are many access control options from the most simple approaches to more complex and flexible XML-based standards. To close out this issue, Eli White has a gloomy forecast for our industry and advice on how to prepare for it in *finally{}: We are in a Tech Recession*

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

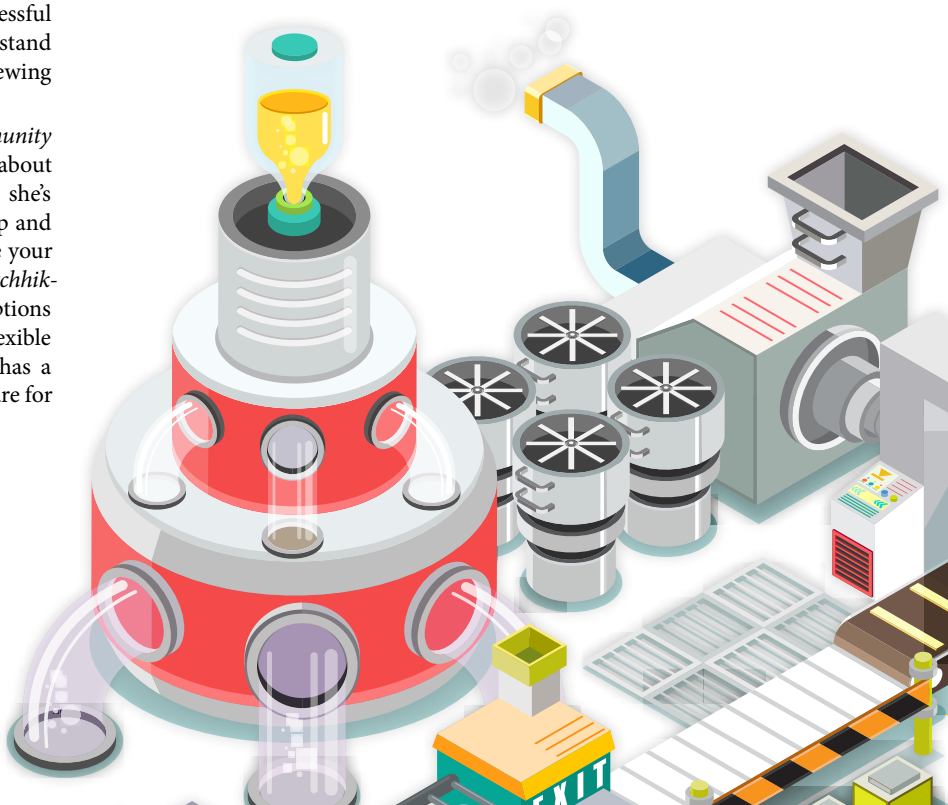
Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us via email, twitter, and facebook.

- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this issue's code package:

http://phpa.me/August2016_code



Deploying with Ansible

Ramon de la Fuente

Ansible is a provisioning tool rapidly growing in popularity, mainly due to its simplicity. But it is capable of more than just provisioning! In this article, I will walk you through building a deploy procedure with Ansible, using the `deploy_helper` module available in Ansible since version 2. Of course, since we are all lazy developers, I will also point you to two existing roles that you can use so you don't have to write your own.

The Problem

A couple of years ago at my company, it was becoming a problem to have different deploy procedures across different projects. We wanted to have a deploy procedure that could be run with a single command and zero manual steps. But deploy procedures are always changing, just like any other piece of code. And when we tried a more complex solution, we discovered that not every developer was comfortable changing the deploy procedure, resulting in unnecessary errors and delays.

Besides being fully automated, we wanted something that could be understood and maintained by everyone on the team. The learning curve should be really low. Finally, porting the procedure to other projects should take as little effort as possible.

Why Ansible

We chose Ansible[1] mainly for its simplicity. This article does come with a bit of a disclaimer. We already provision all of our development boxes (Vagrant) and production boxes with Ansible. So it was considered a small step to replace our deployment tool—we were using Capistrano at the time—with an Ansible version. In fact, for us, it meant removing an entire language (Capistrano is written in Ruby) from the stack.

How Ansible Works (Extreme Quick Start Version)

Using Ansible, you write *tasks* that call *modules* with *parameters*. When executed, Ansible will copy the (Python) module to the remote system, and call it with the provided parameters.

```
- name: This is a task
  git: parameter1=value1 parameter2=value2
  ^ this is the module being used
```

When you are grouping tasks together, you have the option to place them in a *role*. A role is a self-contained set of tasks, default values for variables, and perhaps files and templates. A role exists as a separate folder, by default in the same directory as your *playbook*.

```
├──── role_that_does_one_thing
├──── role_that_does_another_thing
└──── playbook.yml
```

A playbook is the description of everything that you want Ansible to do. For the deploy procedure, the interesting sections are *vars*, *tasks*, and *roles*. You can also add tasks to run before and after the roles (see Listing 1).

LISTING 1

```
01. - name: This is a playbook
02.   hosts: production
03.   remote_user: deploy_user_username
04.
05.   vars:
06.     variable1: value1
07.     variable2: value2
08.
09.   pre_tasks:
10.     ...
11.
12.   roles:
13.     - role_that_does_one_thing
14.     - role_that_does_another_thing
15.
16.   post_tasks:
17.     ...
```

Now, let's take a look at what we want Ansible to do for us.

What Is a Deploy?

Our definition of a **deploy** is a process divided into the following 5 steps:

1. Update the codebase + configuration
2. Install dependencies
3. Preserve shared resources
4. Perform build tasks
5. Finalize

> **Note** *For this article, I'm assuming that the user account used to deploy , as well as any required privileges, have already been created during a provisioning phase.*

1 *Ansible 2:* https://www.ansible.com

A `release` is a complete build of the application being deployed.

Consider the following directory structure:

```
├── releases
│   ├── 20160407234508
│   └── 20160415235146
├── shared
│   ├── sessions
│   ├── source
│   └── uploads
└── current -> releases/20160415235146
```

The `releases` folder contains all of the different versions that have been deployed to this location. In this example, a release is put into a folder named with a date and a time. Having timestamped folders is one way to have distinct releases, but you could choose your own strategy, such as using git tags or commit hashes.

The `shared` folder is for anything that needs to survive after a new release—in other words, anything that is *shared* between the different releases. Examples of this are session files, or files uploaded by users of your application.

The `source` folder contains the source code of the project. The only reason to have it there is to speed up the deploy procedure by not having to download the entire thing every time. This folder is updated during a deploy and then used to copy to a new release folder.

We use a symlink named `current`, which points to the currently active release. A web server uses the path to `current` (or a subdirectory) in its configuration as the web root.

If everything goes well during a deploy procedure, the `current` symlink will be replaced by a link to the newly deployed release. The downtime when switching to a new release is reduced to the time required to switch the link. The directory structure then becomes the following:

```
├── releases
│   ├── 20160407234508
│   ├── 20160415235146
│   └── 20160425170730
├── shared (unchanged)
└── current -> releases/20160425170730
```

Now, let's take a closer look at those five steps.

1. Update the Codebase and Configuration

In the first step, you update the `source` folder to contain the correct version of the project you are deploying. The version is most likely a git reference like a tag or a branch. It should probably be a parameter to the deploy script you use to perform a deploy.

Besides the correct version of the code, you will also need a correct version of configuration files for the environment (development, staging, production, etc.) you are deploying.

Likely candidates for these tasks are the Ansible `git` and `template` modules.

```
- name: Clone project files
  git: repo={{ git_repo }} dest=/path/folder version=v1.0.0

- name: Copy project templates
  template:
    src: prod.yml.j2
    dest: releases/20160425170730/app/conf/prod.yml
```

2. Install Dependencies

The installation of dependencies is usually handled by a dependency manager. In the case of a PHP project, that is Composer, but some projects might also use NPM and/or Bower, for example. Alternatively, you could just download code without the use of a dependency manager with the `url` module.

Since Ansible comes with a `composer` module (but also `npm`, `bower`, `bundler`, and more), installing dependencies could be as easy as a single task:

```
- composer:
    command: install
    working_dir: /path/to/project
```

3. Preserve Shared Resources

In this step, you link the project version you are currently deploying to the resources in the `shared` folder. After this, the new version will have access to the same shared files which the current version does.

The are more ways to make shared resources available in your application, of course, but we use symlinks to shared folders. Those folders already exist in the codebase, and their contents are ignored in git.

This means that it takes a couple of tasks to make the symlink to a shared folder. You could use `with_items` in Ansible to loop over a list of shared resources, but here are the three tasks involved (see Listing 2).

LISTING 2

```
01. - name: Ensure shared/sessions is present
02.   file:
03.     path: 'shared/sessions'
04.     state: directory
05.
06. - name: Ensure app/sessions is absent
07.   file:
08.     path: 'releases/20160425170730/project/app/sessions'
09.     state: absent
10.
11. - name: Link app/sessions to shared/sessions
12.   file:
13.     path: 'releases/20160425170730/project/app/sessions'
14.     src: 'shared/sessions'
15.     state: link
```

4. Build Tasks

Build tasks consist of tools or binaries that you need to run to prepare your code for production use. This includes minimizing JavaScript, compiling Less or Sass, and generating a cached version of the DI container, for instance. As you can see, this varies a lot between projects.

Here is an example task that runs some basic Symfony console commands:

```
- name: Run build commands in the new_release_path
  command: "{{ item }}" chdir=releases/20160425170730
  with_items:
    - "app/console cache:clear"
    - "app/console assets:install"
    - "app/console assetic:dump"
  environment:
    SYMFONY_ENV: "prod"
```

5. Finalize

If you reach this step, everything went well. You are ready to move the `current` symlink over to the new version. There might also be additional things to do, like reloading PHP-FPM or Nginx, or maybe restarting certain long-running jobs that still have the old code in memory.

How can Ansible make your life even easier? I'm glad you asked.

The deploy_helper Module

In order to help you build an Ansible role for all of these different steps without a lot of boilerplate tasks, a module called `deploy_helper`[2] was developed. It is shipped with Ansible 2 by default in Ansible Modules Extra, meaning it is available out of the box.

> *Full disclosure: I wrote this module together with @jaspern-brouwer*

From the manual:

> *The Deploy Helper manages some of the steps common in deploying software. It creates a folder structure, manages a symlink for the current release, and cleans up old releases.*

You run it like any other module by putting it in a task in a playbook:

```
- name: Deploy module
  deploy_helper:
    path: "/home/deploy/projectfolder"
    state: "present"
```

The complete list of parameters can be found in the Ansible manual, of course, but for quick reference, you can check out Listing 3.

Once you run it with `state=query` or `state=present` (the latter is the default), it will create the basic directory structure for you in the `path` you specified. It will also set a few variables—or `facts` in Ansible speak. You can access them in any following task like this:

```
{{ deploy_helper.new_release }}
```

A complete list of the facts and example values (truncated):

```
    project_path: "/home/deploy/projectfolder",
    current_path: "/home/deploy/projectfolder/current",
    releases_path: "/home/deploy/projectfolder/releases",
    shared_path: "/home/deploy/projectfolder/shared",

    new_release: "20160522172756",
    new_release_path: "/hom...eleases/20160522172756",
    previous_release: "20160425170730",
    previous_release_path: "/hom...eleases/20160425170730",

    unfinished_filename: "DEPLOY_UNFINISHED"
```

Most of these are pretty self-explanatory. They provide an easy way to write tasks using the directory structure.

The `new_release` fact is a timestamp generated by the deploy module. If you have a different strategy for naming the folders (like commit hashes or a semantic version), then you can set the `release` parameter explicitly when calling the module. Since the module does not deploy anything, that will only affect the generated facts for use later on.

LISTING 3

```
01. path:
02.   required: True
03.   aliases: ['dest']
04.   description: the root path of the project.
05.
06. state:
07.   required: False
08.   choices: [ present, finalize, absent, clean, query ]
09.   default: present
10.   description: the state of the project.
11.
12. release:
13.   required: False
14.   default: None
15.   description: the version that is being deployed
16.
17. releases_path:
18.   required: False
19.   default: releases
20.   description: the folder that will hold releases
21.
22. shared_path:
23.   required: False
24.   default: shared
25.   description: the folder that will hold shared resources
26.
27. current_path:
28.   required: False
29.   default: current
30.   description: the symlink to a build after finalize
31.
32. unfinished_filename:
33.   required: False
34.   default: DEPLOY_UNFINISHED
35.   description: file to indicate a deploy in progress/failed
36.
37. clean:
38.   required: False
39.   default: True
40.   description: whether to run clean procedure on finalize
41.
42. keep_releases:
43.   required: False
44.   default: 5
45.   description: number of old releases to keep when cleaning
```

The `unfinished_filename` fact also needs some explanation. Because of the way Ansible works, a failed deploy will result in a half-finished build in the releases folder. This is great because it allows you to inspect the contents of the folder and determine the cause of the failure, but it is also a problem when you do not know which build in the `releases` folder is a successful one.

To distinguish between successful builds and unfinished ones, a file can be placed in the folder of the release that is currently in progress. The existence of this file will mark it as unfinished and allow the `deploy_helper` module to remove it during cleanup.

A cleanup is performed by default when you run the `finalize` step:

```
- name: Deploy module
  deploy_helper:
    path: "/home/deploy/projectfolder"
    version: "{{ deploy_helper.new_release }}"
    state: "finalize"
```

The cleanup action will remove unfinished builds in the releases folder, as well as successful builds from the past. You can tell

2 `deploy_helper`: http://phpa.me/ansible-deploy-helper

it how many historical builds to keep by passing a parameter `keep_releases`, but the default is 5 (not counting the current build).

The cleanup action has a safeguard, so it will never remove the active build linked to by the `current` symlink.

With the module at your disposal, it should be pretty straightforward to write a deploy role to fit your project. But as I said before, we are (or at least should be) lazy enough to search for existing code!

If nothing else, existing roles will give you a good place to start for inspiration.

Discovering Existing Roles

Ansible Galaxy

Ansible Galaxy is like Packagist for Ansible roles. You can write and share roles that are then easily added to multiple projects without duplication. It so happens that there are already some that might fit your needs…

https://galaxy.ansible.com/list#/roles?autocomplete=deploy

I will cover two of those here. One is `f500.project_deploy`[3], the role we built and still use for every new project.

The other is `carlosbuenosvinos.ansistrano-deploy`. This is a great example of a similar approach but with some other options and strategies.

The way you get this code into your project is to use the `ansible-galaxy` command. While it is possible to install roles one by one, I would advise you to add a 'requirements file' to your project, so the dependency is documented. The file has a really simple format for roles that come from Galaxy:

```
- src: f500.project_deploy
```

More advanced options can be found in the Ansible manual. Downloading the roles is then as easy as the following:

```
$ ansible-galaxy install -r your-roles-file.yml -f
```

The `-f` or `--force` flag is added because Ansible Galaxy doesn't know how to update (yet)

Project Deploy Example

When using the `project_deploy` role, all you need to do is set the correct variables for your project. It takes care of updating the code through `git` or `rsync`. It also takes a list of templates for your configuration files, and it can do `composer`, `npm`, or `bower` installs. Finally, it takes a list of shared folders to make links to, as well as a list of additional build steps.

One of the smallest possible playbooks looks something like this:

```
- name: Deploy the application
  hosts: production
  remote_user: "deploy"

  vars:
    project_root: "/home/deploy/projectfolder"
    project_git_repo: "git@github.com:F500/some-project"
    project_deploy_strategy: git

  roles:
    - f500.project_deploy
```

3 *Deploy role in Ansible Galaxy:* https://galaxy.ansible.com/list#/roles/732

Admittedly, this does not help you much, as it only takes care of the directory structure and updates the code by putting the "master" branch (the default for the `project_version` variable) in the latest release.

If your project uses `composer`, for example, you need to enable that. And you also have the option to copy the vendor folder from a previous release to speed up the process:

```
project_has_composer: yes
project_copy_previous_composer_vendors: true
```

Adding a template and shared folders is also a matter of setting variables. You can take a look at the entire playbook for the SweetlakePHP website in Listings 4 & 5.

You can see almost everything is done with variables, except a special set of tasks in a file called `finalize.yml`.

When things do not exactly match, you will have additional tasks to run during the different parts of the deploy procedure. The `project_deploy` role helps in that regard by including task files (called hooks) at certain points.

The hooks are the most important part of tailoring a deploy procedure to a specific project. You write your own `some-hook.yml` file and set a specific variable to point to that file. The role then includes the file and runs the tasks:

```
vars:
  project_deploy_hook_on_finalize: "/path/some-hook.yml"
```

The available hooks are as follows: *initialize*, *update_source*, *create_build_dir*,*perform_build*, *make_shared_resources*, and *finalize*.

A very useful Ansible variable for the hooks is `{{ playbook_dir }}`, so you can put your hooks in a folder relative to where your playbook file lives.

LISTING 4

```
01. ---
02. - name: Deploy the application
03.   hosts: production
04.   remote_user: "{{ production_deploy_user }}"
05.
06.   vars:
07.     project_root: "{{ sweetlakephp_root }}"
08.     project_git_repo: "{{ sweetlakephp_github_repo }}"
09.     project_deploy_strategy: git
10.
11.     project_environment:
12.       SYMFONY_ENV: "prod"
13.
14.     project_shared_children:
15.       - path: "/app/sessions"
16.         src: "sessions"
17.       - path: "/data/uploads"
18.         src: "uploads"
19.
20.     project_templates:
21.       - name: parameters.yml
22.         src: "templates/parameters.yml.j2"
23.         dest: "/app/config/parameters.yml"
24.
25.     project_has_composer: yes
26.     project_copy_previous_composer_vendors: yes
27.
28.     project_deploy_hook_on_finalize: \
29.       "{{ playbook_dir }}/finalize.yml"
30.
31.   roles:
32.     - f500.project_deploy
```

finalize.yml **LISTING 5**

```
---
- name: Clearing the cache
  command: app/console cache:clear
  args:
    chdir: "{{ deploy_helper.new_release_path }}"
  environment: "{{ project_environment }}"

- name: Migrating DB
  command: >
    app/console doctrine:migrations:migrate
    --no-interaction
  args:
    chdir: "{{ deploy_helper.new_release_path }}"
  environment: "{{ project_environment }}"

- name: Installing assets
  command: app/console assets:install
  args:
    chdir: "{{ deploy_helper.new_release_path }}"
  environment: "{{ project_environment }}"

- name: Assetic dump
  command: app/console assetic:dump
  args:
    chdir: "{{ deploy_helper.new_release_path }}"
  environment: "{{ project_environment }}"
```

By default, the `project_deploy` role will end the deploy procedure by replacing the `current` symlink. If you do not want the deploy role to do that just yet, you can set the variable as `project_finalize: false`. This allows you to do potentially destructive things like database migrations *after* the deploy role finishes. The symlink can still be replaced later with a one-liner calling the `deploy_helper` module with `state=finalize`.

Ansistrano Example

Using Ansistrano[4] is very much like the `project_deploy` role. I'll highlight some of the differences:

- The variables are named differently (of course).
- It does not use the `deploy_helper` module (although the directory structure is very similar).
- It does not assume a dependency or package manager, so there is a little more work to do when you want do a `composer install`, for example. There is also no default way to add templates. But because of a similar "hooks" mechanism, it is very easy to do.
- The available hooks are as follows: *before/after_setup_tasks_file*, *before/after_update_code_tasks_file*, *before/after_symlink_shared_tasks_file*, *before/after_symlink_tasks_file*, *before/after_cleanup_tasks_file*.

The Ansistrano role has a companion called `ansistrano_rollback` that can replace the symlink from the current release back to the release before. This makes it easier to undo a botched release. Very neat!

Ansistrano has a larger community following and is quickly becoming the most popular role *in the entire Galaxy*.

Take a look at the example in Listing 6 to get an idea of the difference between the two roles. It took about 30 minutes to start using the Ansistrano role to deploy SweetlakePHP.

LISTING 6

```
---
- name: Deploy the application
  hosts: production
  remote_user: "{{ production_deploy_user }}"

  vars:
    project_environment:
      SYMFONY_ENV: "prod"

    ansistrano_deploy_to: "/home/deploy/ansistrano"
    ansistrano_deploy_via: "git"
    ansistrano_git_repo: "{{ sweetlakephp_github_repo }}"

    ansistrano_shared_paths:
      - "app/sessions"
      - "web/uploads"

    ansistrano_allow_anonymous_stats: no
    ansistrano_keep_releases: 5

    ansistrano_after_update_code_tasks_file: \
      "{{ playbook_dir }}/ansistrano-after-update-code.yml"
    ansistrano_before_symlink_tasks_file: \
      "{{ playbook_dir }}/ansistrano-before-symlink.yml"

  roles:
    - carlosbuenosvinos.ansistrano-deploy
```

ansistrano-after-update-code.yml **LISTING 7**

```
---
- name: Deploy the application
  hosts: production
  remote_user: "{{ production_deploy_user }}"

  vars:
    project_environment:
      SYMFONY_ENV: "prod"

    ansistrano_deploy_to: "/home/deploy/ansistrano"
    ansistrano_deploy_via: "git"
    ansistrano_git_repo: "{{ sweetlakephp_github_repo }}"

    ansistrano_shared_paths:
      - "app/sessions"
      - "web/uploads"

    ansistrano_allow_anonymous_stats: no
    ansistrano_keep_releases: 5

    ansistrano_after_update_code_tasks_file: \
      "{{ playbook_dir }}/ansistrano-after-update-code.yml"
    ansistrano_before_symlink_tasks_file: \
      "{{ playbook_dir }}/ansistrano-before-symlink.yml"

  roles:
    - carlosbuenosvinos.ansistrano-deploy
```

4 *Ansistrano:* http://ansistrano.com

Related Topics

Ansible Vault

When you deploy your projects, the configuration files will be different per environment. And when you start sending code to production, you need to send the production credentials along with that. This becomes a security hazard if you add passwords to the deploy code and add that in the repository. This is not just for production passwords, though—your staging/testing credentials and external API keys fall into the same category.

> *Please, please do not add plaintext credentials to your repo, even if it is "private."*

In order to safely add credentials, Ansible allows you to encrypt files using the `ansible-vault` command. This allows you to enter a password, and it will encrypt the entire file. It pays to separate the non-secure variables from the secure ones, though, so you keep the benefit of version control and human readability for anything that does not need to be encrypted.

When you run a playbook that tries to access an encrypted file, you have to add the `--ask-vault-pass` parameter or else the process fails. You then give the password to `ansible-playbook` at runtime so it can decrypt your passwords.

Maintenance Mode

There is no general concept of "maintenance mode." It depends very much on the tools that you use, but it is generally a good idea to have some tasks that enable/disable access to the application while you are performing the version switch.

This also includes possible long-running jobs that need to be restarted or cron jobs that might be activated at an inappropriate time, such as in the middle of a deploy.

For example, we are currently writing a file to the file system that is used in Nginx configuration and console commands. If the file exists, requests immediately return with an "in maintenance" response without hitting the application. Simple but effective.

Database Migrations

It pays to automate your schema changes with a tool like Doctrine Migrations or Phinx. Even if you write the migrations by hand, you are documenting the changes to your database and making it easy to update many different (including local) environments.

When you have a reliable process in place, it's trivial to add that to your deploy procedure. Being the astute readers you are, you probably all noticed the following task in Listing 2:

```
- name: Migrating DB
  command: >
    app/console doctrine:migrations:migrate
    --no-interaction
  args:
    chdir: "{{ deploy_helper.new_release_path }}"
  environment: "{{ project_environment }}"
```

This updates the database, but it does so *before* the symlink is replaced. This is what we call `fingers crossed deployment`.

I can really only recommend this practice in two cases. The first is if you are an extremely lucky person. The second is when you have absolutely no visitors on your website, which is the case with SweetlakePHP…

Fingers Crossed deployment

All jokes aside: doing it this way is a risk. Don't say I did not warn you!

Rollbacks

A *rollback* is the concept of turning back steps in a deploy when something goes wrong. Ansible is not a programming language, and as such, it is very hard to "jump" to a specific part of the playbook or undo certain actions.

Because of that, we prefer a "fail forward" mentality. Stuff *will* break, but instead of always trying to move back to a previous state, it's often easy enough to fix the problem and deploy again.

In Ansible 2.0, a new concept was added that looks a lot like `try {} catch {} finally {}`. It is written as:

```
- block:
    # do something
  rescue:
    # do something on error
  always:
    # always do something
```

This new concept will be useful when writing deploy procedures, although at the time of this writing, I did not find any examples of deploy roles in the wild.

Wrap Up

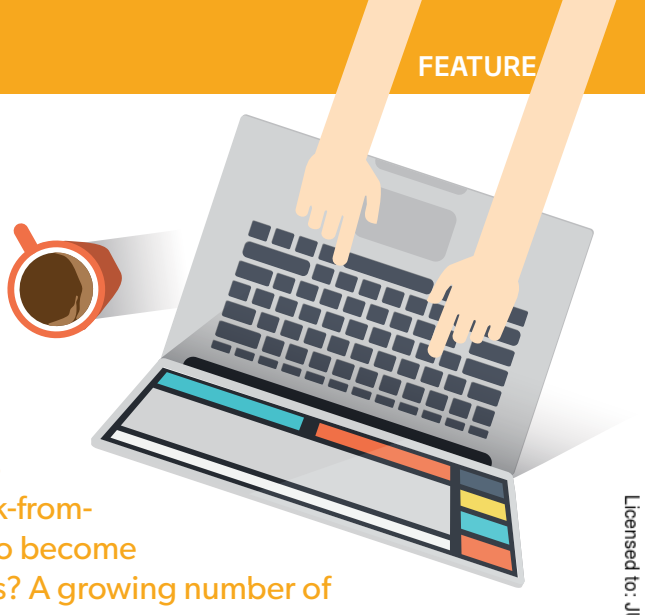
I hope I have shown you enough to start deploying your applications with Ansible. For a standard setup of most frameworks, one of the existing roles should do just fine. If you feel you have to shoehorn your deploy procedure into those roles, just take them as an example, and roll your own! Having a zero-effort deploy mechanism opens the door to automated deployment pipelines and releasing new versions as often as you need to. And those are very powerful assets that will allow your development process to run a lot smoother.

Ramon is a company owner, developer and family man (not necessarily in that order). With 16+ years of experience in web development, he has seen technologies come and go—but robust design is always a must. *@F_U_E_N_T_E*

Further Reading:

- Thoughts on deploying with Ansible: http://phpa.me/thoughts-deploying-ansible

More Than Just an "OK" Dev



Gabriel Somoza

In today's web development market there are tons of job openings with excellent salaries and perks like cars, work-from-home arrangements, unlimited holidays, opportunities to become certified, and so on. But how long until the bubble bursts? A growing number of people are being raised to become programmers, and some have been programming since before they could even talk. Are you ready to compete with the upcoming waves of younger, smarter developers?

I was raised in Argentina, one of a few countries where you wouldn't necessarily be surprised to learn that your taxi driver had a medical degree. As in many other parts of the world, doctors in Argentina are overworked and underpaid.

At the other end of the spectrum, we developers experience almost the exact opposite. Companies are competing very aggressively to attract talent, to the point of basically treating developers like rockstars—giving them beautiful company cars, unlimited holidays, remote work arrangements, and even four-day work weeks, all on top of an average U.S. salary of nearly six figures—and well above that in some other places.

When considering the contrasting working conditions of these two professions, one thing is clear: we're living in the golden age of web development. In the words of Silicon Valley developer James Somers[1]: "In this particular gold rush (of tech startups) the shovel is me."

Such market conditions can foster developer complacency and professional stagnation. For example, a developer might stay long-term at a comfortable and well-paid job that's not challenging enough for his or her career, and could simply lose the drive to learn new things.

I regularly hire developers and frequently encounter applicants who seem content to stagnate. As I interviewed people, I learned to quickly identify certain characteristics that, in most cases, make all the difference between someone being an "OK" developer and an outstanding one. I have listed some of them below, along with some advice on how to build them into your career toolbox.

1 *Web Developer: It's A Little Insane How Well-Paid And In-Demand I AM:* http://phpa.me/bi-webdev-demand

1. Engages with the Community

This refers to developers who regularly attend meetups or conferences and, usually, are active on IRC, Slack, and other online forums.

Ideally, the developer is a community **enabler**: actively helps organize events, regularly speaks at meetups or conferences, and provides support in online technical communities. A community enabler can communicate and solve complex interpersonal problems beyond the realm of code.

These traits are valuable because communities are bigger than individuals. A developer actively involved in the community is connected to something larger than himself and will therefore have continual access to:

1. New ideas, which can prevent stagnation.
2. Other experts and specialists, including the problem-solving resources and potential new hires they can supply. Such value is often overlooked.
3. Mentors.

Starting to Participate in the Community

The development community will gladly help you to integrate if you show interest. You can start by joining a local meetup (try http://meetup.com). Twitter, IRC, and Slack are great venues when you're having trouble or if you're willing to grow by helping others.

2. Contributes to Open Source

Open source is the heart of PHP, and I've come to realize that great PHP developers understand this. Contributing to Open Source projects is therefore one of the best ways to give back to the community.

I now rarely consider hiring someone for a senior position if that person doesn't have publicly available contributions, even just a personal project. The project must be complete, and, ideally, it should be a reusable library that solves a well-defined problem.

When reviewing a developer's public code, it's important to look for the basic indicators of a good developer: automated tests, a well-defined coding style, documentation, design, license inclusion, proper use of patterns, proper use of Composer (including a good choice of dependencies), and so forth.

A developer who regularly contributes to open source displays the following characteristics:

1. experience designing code for reusability
2. self-motivation and proactivity
3. attention to detail

Other ways of giving back to the community, such as mentoring, will be discussed later in this article .

Starting to Contribute to Open Source

When I wanted to start contributing, many people told me that I simply had to submit a pull request. My problem was not knowing *what* to contribute. Eventually, I realized that getting started is not so much about chasing a great idea as it is about coding with a specific *mindset* of solidarity: "Will this code I'm writing be valuable to the community?" The answer is probably **yes** if you:

1. Fixed a bug
2. Found a better way of doing things (something faster, less resource-intensive, more flexible, etc.)
3. Found something that your framework was missing. For example, one of my first pull requests was a Whitelist and Blacklist filter for Zend Framework 2.
4. Created something modular that can be reused with a framework, or even by itself.
5. Helped to document a framework or library
6. Added automated tests for a framework or library

As you explore the mindset and become familiar with the above, you'll find many other opportunities to contribute code, and ultimately give back to the community.

> **TIP #1:** *Before spending a lot of time working on a new feature or fixing a bug on a project, go to the project's issue tracker and make sure there are no open issues related to that.*

> **TIP #2:** *If you're contributing a new feature or a backwards-incompatible change, make sure you open a new ticket on the project's issue tracker to discuss it first, before you spend time working on it. The project maintainer may have different plans in mind, so first make sure your efforts will be well received!*

3. Learns from Mentors

> *A mentor is a person or friend who guides a less experienced person by building trust and modeling positive behaviors. An effective mentor understands that his or her role is to be dependable, engaged, authentic, and tuned into the needs of the mentee*[2].

It's relatively common for people to find a mentor because it's natural for human beings to transmit knowledge from generation to generation. Many people have one (or more) mentors during their careers without even realizing it: a teacher at school, a family member, a colleague or boss, or even a consultant working at their company.

When I look back I realize my career wouldn't be the same if it wasn't for the mentors I had.

For me, it started early: when I was around 11 years old my computer teacher saw that I had an interest in programming. While most other kids were learning how to type without looking at the keyboard, he would invest time to show me how to build a few things in Macromedia Flash 5 (ActionScript). That's how I started to build my first Flash games and websites, and how I developed the passion for web development that I still have today. Since then I've been lucky to always have at least one mentor in my life. Eventually it became a conscious choice: I started to actively look for mentors to coach me on several aspects of my career.

One of the main reasons I search for developers who have mentors is that I know I'll be hiring someone who has direct access to a pool of knowledge and experience that's larger than the individual. This is similar to the concept of being tapped into a community, except that in this case it carries the stronger bond of a deeper relationship.

Finding a Mentor

There are many paths to finding a good mentor, but in my experience the best mentoring relationships happen **organically**. The most important ingredients that help these organic relationships to develop are being involved in the community and giving back to it—which, not by coincidence, are the first two items discussed in this list. While doing that you'll cross paths with people that you'll start to admire.

Once you find the right person, the next step is formalizing the relationship. There are many articles online on how to engage with a potential mentor and ultimately define and formalize the relationship.

Last but not least, in the PHP community there's a project at http://php-mentoring.org specifically designed to help people find mentors. The site also has more information about the benefits of mentoring as well as a directory of people who are willing to mentor. I encourage you to check it out!

Becoming a Mentor

As I developed my career I had the opportunity to mentor developers myself, and found that being a mentor is an excellent way to learn even more. In other words, it's a two-way street for both the mentor and the mentee.

Becoming a mentor is a bit more difficult than finding one. though, because you have to be approached, which requires a certain level of recognition and respect. But once again the key ingredient is community: become a community **enabler**, be ready to help, lead by example, always while being respectful of others, and you'll soon become someone's role model.

4. Is a Full-Stack Integrator

This is something I realized only recently: the value of the "full stack developer" title is getting diluted because, as technology evolves, most levels of the stack are becoming highly specialized. It's not only hard to become a full-stack developer: it's *even harder* to maintain the requisite knowledge. It is therefore becoming increasingly rare to find someone who truly qualifies as a full-stack developer.

Instead of looking for full stack developers, I look for people who qualify as **full-stack integrators**. These people may specialize in an area such as back-end PHP development, but they have enough end-to-end architectural skills to integrate new technologies fairly easily, even without extensive knowledge. For example, they may be able to quickly build an API to communicate with an Ember.js front end, even without having worked with Ember.js before. They may even be able to change existing Ember.js code without spending much time reading documentation.

Some key characteristics of full-stack integrators:

2 *Module 3: What is a Mentor and Roles of the Mentor and Mentee: http://www.oycp.com/MentorTraining/3/m3.html*

1. They can filter noise. This allows them to skim documentation very quickly and extract only the information they really need to solve the problem at hand.
2. They have 90/10 skills, meaning that they're skilled with one specific technology (the 90 part), but they also have basic skills and experience in a wide range of other technologies across most levels of a modern web stack.
3. I've found that these people usually market themselves as specialists, not as full-stack developers. However, their job applications and public code make them stand out from pure specialists by showing that they're not afraid to get their hands dirty with new technologies.
4. They share a deep passion for technology and rarely say things like "technology X is the best"; they know when to apply the right technology.
5. Many of them are either leading development teams or running their own companies, which allows them to practice their skills as needed.

Becoming a Full-Stack Integrator

Becoming a full-stack integrator requires many years of experience integrating, designing, and delivering solutions with different technologies. Most of the people I've worked with who have this experience have pivoted their web stack at least once (usually multiple times) during their careers. For example, they went from being a senior Rails or Java developer to a NodeJS developer, then moved to PHP.

Becoming a full-stack integrator is also supported by a shift in paradigms: OOP to functional, monolithic to micro-services, CRUD to DDD, etc. This provides deep architectural understanding, which is essential for successfully designing and integrating an application's technology stack.

5. Follows Coding Standards and Best Practices

A typical code review involves looking at the cleanliness of the code, the extent to which it's documented, the presence of tests, and so on. Looking under the hood, many of these indicators are related to how much a developer respects code—not only her own but also that of others—to the extent that she will follow a set of coding standards and best practices.

Coding standards offer several benefits to adopters: sustainability, interoperability, reduced cognitive overhead, reduced risk, improved productivity, and more. Some have become so widely adopted in the PHP community that they're indispensable for modern web development: a clear example is PSR-4 (the Autoloader standard). They therefore reflect some of the things that the community, company, or project behind them finds important—such as the need for better interoperability between frameworks in the case of the standards recommended by the PHP-FIG[3]. Although adhering to them can be beneficial at a project level, it is ultimately also a way of showing empathy and respect to the community that recommends them.

When hiring senior PHP developers I consider adherence to major standards and best practices a strict "must" (except when properly justified). I've found they can work as great filters during interviews: if the person can't tell me what the first 4 PSRs are about without googling it, then they're immediately discarded from the process. The same thing happens if their open source code doesn't follow a clear coding standard.

All great PHP developers I know of follow coding standards (such as PSRs or PEAR) and best practices (such as writing automated tests). And if they don't follow one of them, they usually **document** an excellent reason why, providing useful information for future developers in the project.

Learning About the Latest Standards and Best Practices

My favorite go-to resource for learning about the latest PHP development standards and best practices is the PHP The Right Way website[4]. If you haven't visited it yet I strongly encourage you to do so—the site includes many other resources and links (mostly technical) that will help you become a better developer.

If you already know about these standards and best practices but you feel you don't have a deep grasp of the subject yet, the best way to cement your knowledge is to apply it. Next time you work on a project, for example, make sure it has the following elements:

1. A `CONTRIBUTING.md` file (or equivalent) that explains in detail which standards the project adheres to.
2. A test suite that includes static code analysis to enforce those standards. You can use CodeSniffer[5] and PHP Mess Detector[6] to help you with that.
3. A build tool to automate those tests, such as Phing[7] or Ant[8].

> **TIP:** *You can contribute a pull-request to any open-source package that doesn't do this yet, as a way of giving back to the community!*

I find there have been few things more satisfying in my career than the opportunities I've had to learn technologies and get involved with the communities around them; and there's been nothing more inspiring than following the example of those ahead of me. Discovering the value of community has also refueled my passion for web development in moments when it was dwindling, and I know it has the potential to do so for many years to come.

Conclusion

To conclude, we briefly examined some characteristics that can help you stand out beyond your code. Even better: they will help you stand out *within the community*—with all of the benefits and responsibility that entails. As you might have realized, we barely scratched the surface on each of them—and there are many more—so I encourage you to pick one, start to develop it today, and learn more as you go!

Gabriel Somoza is a Belgium-based PHP architect with a strong background in Ecommerce. He spends most of his time consulting through Strategery—his development boutique specializing in building highly-customized web applications and e-commerce stores. He's also a ZCE and Magento Certified Developer (Plus). He is active in the open-source community with contributions to Doctrine Migrations, ZF2, and is the author of the Strategery InfiniteScroll extension for Magento. He's the organizer of PHP Limburg, a meetup with 120+ registered members in Belgium. When not working, he spends time with his beautiful wife, travels, and researches new ways of applying technology to solve old problems.

3 *PHP-FIG:* http://www.php-fig.org

4 *PHP The Right Way website:* http://www.phptherightway.com

5 *CodeSniffer:* https://github.com/squizlabs/PHP_CodeSniffer

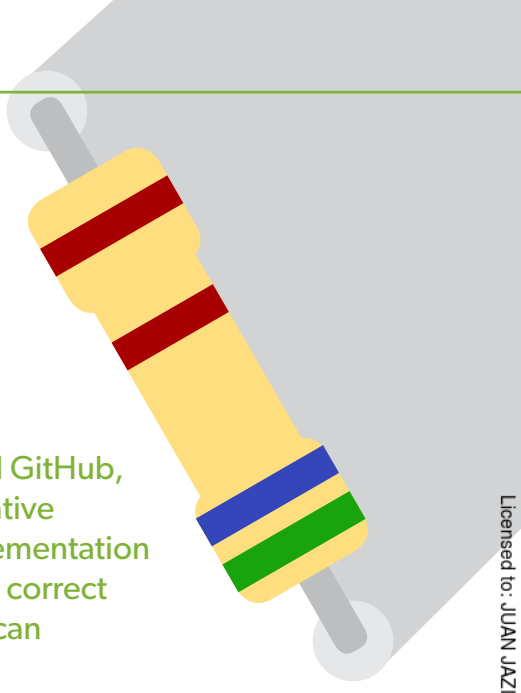
6 *PHP Mess Detector:* https://phpmd.org

7 *Phing:* https://www.phing.info

8 *Ant:* http://ant.apache.org

What's Your Code Tolerance?

Steve Bennett



Thanks to feature-rich source code management software like Stash and GitHub, peer code reviewing is no longer the painful task it once was. Collaborative workflows pair with source code management software to aid the implementation of best practices during the peer code reviewing process. Applying the correct amount of tolerance during peer code reviewing is a best practice that can significantly impact the health of your project and sanity of your peers.

Framing Tolerance

Day 1 of my high school electronics course was spent learning about Ohm's law[1] (I = V/R). Ohm's law states that "the current through a conductor between two points is directly proportional to the voltage across two points." To put another way: it describes how voltage, that thing that can make you say "ouch" when you touch it, changes as it encounters resistance flowing through a circuit.

Typically, the resistance voltage encounters IS intentional. It is put in place to help reduce the voltage to a point that, say, a computer chip, can operate. Intentional resistance is achieved by using resistors. Work with resistors long enough and you'll find that most come color-coded. Resistors with a five-band color code use the fifth band to indicate the tolerance of the resistor. Yes, I know, we're finally getting to the point.

Let's say we have a resistor that's rated at 1K Ohms, the Ohm being one unit of resistance. If our resistor has a tolerance of plus or minus 1 percent (tolerance is represented as a percentage), you can expect that the amount of resistance the resistor will provide will be between 990 Ohms and 1010 Ohms. OK, no need to discuss further, just hang on to that while you continue to read.

Collaborative Workflows and Tolerance

With the widespread adoption of Git and feature-rich source code management (SCM) software like Atlassian's Bitbucket and GitHub, it's now easier than ever to implement peer code reviewing into our everyday development process. Collaborative workflows can make the peer reviewing of code something that's no longer feared. Practice it long enough and it becomes second nature, just another step in the development lifecycle. The benefits of peer code reviewing are numerous and include fewer application bugs, more stable code, better application architecture, increased project communication and more knowledgeable developers, to name a few.

Hopefully you're reading this and thinking to yourself that your development team already practices peer code reviewing and some form of a collaborative workflow. Unfortunately, there are still many large development teams out there using—and I'm about to say a bad word here so cover your ears—*Subversion (SVN)*, or another similar archaic VCS (version control system). I'm giving SVN a hard time, and please don't be offended at my comic relief if you're still using it as your primary VCS. I stated there are still large development teams using SVN and that comes from personal experience working with those teams. Within the last year our team has worked with and migrated two large corporations from existing Subversion projects to Git. Many times, organizational structures or simply fear of migration cost and team adoption will get in the way of moving off SVN.

I'm hitting SVN hard here because I strongly believe it's not possible to implement an effective collaborative workflow without a proper VCS foundation. SVN, still popular among many development teams, does not allow for a proper collaborative workflow process. The branching mechanism in SVN is ineffective, merge conflicts can take hours to fix, and the list goes on. Our development team will not take on an existing project maintained in a VCS other than Git unless the client is willing to migrate.

The process of migration from Subversion to Git is surprisingly simple. Just as you would expect from the perfect child that Git is, Git provides a tool to assist you with the migration in ***git svn***. There is a plethora of other migration tools on the market to assist with migrations. If you're simply looking to move your Subversion trunk and nothing else, ***git svn*** will likely be sufficient. If you also need to transfer existing branches or maintain a repository sync during a longer conversion timeline, then I would recommend a tool designed around this style of migration. Our team uses the Atlassian suite of products extensively and, as expected, a feature-rich tool to assist with migrations is provided by Atlassian. You can read more on the Atlassian SVN to Git migration tool online[2].

Well, that was a long SVN tangent, but it feels oh-so-good to know that we're now all using Git. Let's start to frame our collaborative workflow / peer review process discussion and apply that to the concept of code tolerance. We'll start by peeling those two concepts apart. A collaborative workflow is the mechanism that allows for proper peer code reviewing. A collaborative workflow can be implemented independently of peer code reviewing. However, proper peer code reviewing cannot be achieved outside the process of collaborative workflows. Correct tuning of these processes into a well-oiled machine is essential to the production of successful projects.

What are collaborative workflows? There are many variations of collaborative workflows out there. As such, the definition will change based on the industry or workflow

1 https://en.wikipedia.org/wiki/Ohm%27s_law

2 *Migrate to Git from SVN:* http://phpa.me/git-svn-migration

software referenced. Our definition of collaborative workflows will focus on workflows as they pertain to Git. Even within the world of Git there are many workflow variations. I will not attempt to declare one flavor as superior to another. Picking a workflow to follow will likely depend on the size of your development team and your level of Git adoption. For the purposes of this article, I'll define a collaborative workflow as the system by which contributions are made to a repository. Simple enough, right?

The workflow our own development team implements is a healthy blend of Feature, Gitflow and Forking, but most closely aligns to that of a Forking Workflow as defined by Atlassian[3]. In the forking workflow, a developer creates a fork or copy of the central repository. The developer then works from their fork of the central repository. This is done by cloning the fork locally for development purposes. He/she will create a local feature branch that contains the "contributions" intended to be merged into the central repository. Note: From this point on we'll use the term "feature branch" to designate the "contributions" being made to a central repository.

In order for the feature branch to be merged, the developer must push the new feature branch to their fork, or copy of the central repository. The developer then creates a pull request for the new feature branch on their fork to be merged into the central repository. The pull request allows the central repository maintainer(s) a chance to review the feature branch created by the developer.

This is the crucial point that can make or break a project. The process of repository maintainer(s) reviewing requested "contributions" made to a central repository is our definition of peer code reviewing. The importance of peer code reviewing cannot be understated. Peer code reviewing is a skill learned by repetition. Its primary function is to ensure the adherence of "contributions" made to the central repository to the defined project specifications and coding standards, but also has many secondary effects on the health of a project.

We've been using the term "proper peer code reviews" for a while and now it's time to wrap it with a definition by way of best practices. Within the context of our own development team, proper peer code reviewing includes the following best practices.

3 *Forking Workflow:* http://phpa.me/atlassian-forking-workflow

Ensure that feature branches (and bugfixes) meet the requirements defined for a given task. The manual side of branch verification is achieved by checking out a local copy of the feature branch to be merged and testing that the functionality provided by the branch matches the requirements specification.

The automated side of branch verification involves implementation of an automated testing environment. Bamboo and Travis CI are a couple of notable options for managing this. There are numerous automated tests that can be run. I would recommend, at the very least, setting up PHPUnit[4] and PHP_CodeSniffer[5] in your project. Both can be composed into a project with minimal effort. There's also nothing for the reviewer to do here other than ensure the automated tests complete/build successfully.

If there's one step to peer code reviewing that is most frequently left out, this is it. I am, as I'm sure you are, guilty of skipping this step more frequently than I'd like to admit. If I had to pick one part of peer code reviewing I would consider most valuable to the health of a project, this is it. It's one of the easiest ways to smoke-test your application before sending it off to the dreadful user acceptance testing phase. It can be used to ensure a developer is properly completing requirements for tasks. It also directly replicates the environment for a reviewer to use when it's time to teach a young Padawan a thing or two.

Assign multiple peer reviewers. This one isn't always possible, as development team sizes vary from project to project and company to company. SCMs will allow assigning of multiple users to pull requests. Approval is required from all assigned reviewers before a pull request for the feature branch can be merged into the central repository. Even if you're soloing the development effort for a project, it's a good idea to assign reviewing authority to a fellow team member. They may not take the time to set up the project and test your feature branch locally, but a second set of eyes reviewing your changes will help ensure you're not slipping on your standards.

Look for the teachable moment. When code falls short of the coding standards or technical expectations set by the reviewer, provide feedback and direction for revisions in a manner that builds the skill set of the reviewee and doesn't degrade the work they've already completed.

4 *PHPUnit:* https://phpunit.de

5 *PHP_CodeSniffer:* https://github.com/squizlabs/PHP_CodeSniffer

Proper peer code reviews use a defined set of coding standards to which the feature branch should adhere. The assigned peer code reviewers should assess the feature branch to ensure adherence to these standards. Github and Atlassian's Bitbucket are two great SCM products for this part of the peer review process. They allow the reviewers to see a visual comparison of differences between the central repository and the feature branch.

Finally, apply the right code tolerance. You'll find as you repeat the process of peer code reviewing, many decisions made around coding standards will fall into a gray area. The code may adhere to the coding standards but fail to meet the technical expectations of the reviewer. It's during this part of the peer code review process that a tolerance decision will need be made.

Applying Tolerance

Developers typically come from different coding backgrounds and will apply their own approach to each development solution. You use a `for` loop in places where I typically use `foreach`. I know, I know, weak example, but we'll let it slide. Now I'm reviewing your code, looking at your new, carefully crafted `for` loop. It's an OK implementation, but I've seen better, and I think my `foreach` would be more aesthetically pleasing. That little twitch in my brain is so strong that I just can't stop myself from commenting on your pull request (PR). Of course, I use my fancy SCM to do so, and wait for you to change the code before I approve it.

My example is a little extreme, but hopefully you see my point. My peer now needs to shift context because they probably saw the email notification pop up while they were working on another project (another feature of our fancy SCM software). He or she calls me to discuss why their implementation of a `for` loop wasn't appropriate. After giving up the debate because, well, I am right about my `foreach`, they change their IDE from the project they were working on and check out the feature branch for the pull request in question. Then they make the requested change, rebase the branch and push the fixed feature branch. They'll also want to open the fancy SCM software and reply to the original comment to notify me that they fixed my twitch.

Remembering our initial framing of tolerance as it relates to resistors, let's look at the manufacturing process for resistors and see what other parallels we can draw. The manufacturing process for resistors works a lot like that of CPUs (central processing unit). It's possible that a 2.5 GHz CPU could be manufactured on the same wafer as a 2.0 GHz CPU. Defects in the manufacturing process cause the difference in processing speed. Manufacturers test the CPUs for their optimal performance and sell them as such. The resistor manufacturing process works in the same way. Resistors are manufactured for a specific Ohm value. Resistors that test above or below the intended Ohm value are marked with a tolerance band to indicate the difference. Just like a higher-speed CPU, lower-tolerance resistors are sold with a higher price tag.

As with resistors, my low tolerance level comes with a cost. The low tolerance level I applied in my "weak" peer reviewing example likely just cost the project a minimum of 15 minutes. Rinse and repeat this over the course of a six-month project. You can see how this starts to add up and why the correct application of tolerance is important. Not only does a low tolerance result in lost project time, it can affect the cohesion of a development team. Apply too little tolerance in your peer reviewing process and you risk burning out the other components.

Of course, the real world isn't as black-and-white as my "weak" example used above. Let's up the complexity of our tolerance decision by considering the code in Listing 1.

There's nothing functionally wrong with the code above. We're injecting a service locator object into our class. We need two services from the locator, so why not just inject the service locator and use it to fulfill both dependencies, right?

Being the all-knowledgeable developers we are, we see some issues with exposing the entirety of the service locator to the `MySampleClass`. We might also be considering some issues trying to unit test this, although the client isn't paying for unit tests so not much impact there. We're now faced with a tolerance decision. Let the code stand or send the developer through the cycle we outlined in our "weak" example?

LISTING 1

```
01. <?php
02. class MySampleClass
03. {
04.     /**
05.      * @var ServiceLocatorInterface
06.      */
07.     private $serviceLocator;
08.
09.     /**
10.      * @param ServiceLocatorInterface $serviceLocator
11.      */
12.     public function __construct(
13.         ServiceLocatorInterface $serviceLocator
14.     ) {
15.         $this->serviceLocator = $serviceLocator;
16.     }
17.
18.     public function myServiceMethod() {
19.         $countryRepo = $this->serviceLocator
20.                             ->get(MyCountryRepo::class);
21.         $languageRepo = $this->serviceLocator
22.                             ->get(MyLanguageRepo::class);
23. // ... Do something with $countryRepository here
24. // ... Do something with $languageRepository here
25.     }
26. }
```

Upping the complexity of our tolerance decision further, consider the following revision to our first complex example code:

There's really nothing wrong with this example. We've removed our dependency on the service locator and injected each repository dependency separately. Though one could still argue that instead of injecting the repositories, you should only inject the result of the getter. In our example, this means only injecting the `$countryCode` and `$language` variables. Again we are faced with a tolerance decision.

The point I'm attempting to convey with my complex tolerance examples is not one of right and wrong. It's a thought exercise in recognizing situations where code tolerance can be applied in the peer code review process. Correct code tolerance application is, like the peer code review process as a whole, a skill built with repetition. The longer you apply tolerance to your peer code review process the easier the decision of how much tolerance to allow becomes.

LISTING 2

```
01. <?php
02. class MySampleClass
03. {
04.     /**
05.      * @var MyCountryRepository
06.      */
07.     private $myCountryRepo;
08.
09.     /**
10.      * @var MyLanguageRepository
11.      */
12.     private $myLanguageRepo;
13.
14.     /**
15.      * @param MyCountryRepository $myCountryRepository
16.      * @param MyLanguageRepository $myLanguageRepository
17.      */
18.     public function __construct(
19.         MyCountryRepository $myCountryRepository,
20.         MyLanguageRepository $myLanguageRepository
21.     ) {
22.         $this->myCountryRepo = $myCountryRepository;
23.         $this->myLanguageRepo = $myLanguageRepository;
24.     }
25.
26.     public function myServiceMethod() {
27.         $countryCode = $this->myCountryRepo->getCountryCode();
28.         $language = $this->myLanguageRepo->getLanguage();
29. // ... Do something with $countryCode here
30. // ... Do something with $language here
31.     }
32. }
```

Before I close this out I'm going to take a paragraph here and defend myself from the inevitable backlash of comments I will surely receive from the ivory towers reading this. I think I've made this clear, but I am in no way suggesting that peer code review is not important or that there are no wrong ways to code things. There are most definitely wrong ways to code things. If a feature branch doesn't work at all or doesn't follow coding standards set in place by your team, then it will need to be fixed. On the other hand, if a feature branch is working correctly but simply doesn't meet the technical expectations of the reviewer, consider upping your tolerance.

Use the pull request for the feature branch as a teaching opportunity for the more junior developer, but allow the feature branch to be merged. You'll save your project development hours while creating less frustrated and more knowledgeable peers.

Building a Tolerance

One last analogy to drive my point home. At *Soliant*[6], we often liken our *Development Process*[7] to that of homebuilding. We start with our foundation phase or blueprinting. We then build a foundation for the application. In our case that foundation is typically Zend Framework[8]. We frame out the application into separate modules, or rooms if you will. Once our framing is complete, we then finish the build according to the blueprints.

During the course of typical home construction there will be hundreds or even thousands of cuts made. Some cuts made by experienced craftsman and some by the high school graduate looking to make money for their freshman year of college. The experienced craftsman knows that when cutting a board to length, one should account for the width of the saw blade making the cut. The high school graduate does not and cuts to the inside of the mark, making the board an eighth of an inch too short.

Now it's time for the experienced craftsman to make a tolerance decision. Use the short board—after all it still fits and will pass inspection. Or scrap the board, reiterate how to make a proper cut, then send the high school graduate back to try again. Each scenario gets the house completed according to building codes and will more than likely not have an effect on the longevity of the home. However, the second option will produce project waste and a loss of time.

The review of the graduate's cut is something that is a necessity. There is no scenario where the process of peer reviewing can be removed. The underlying issue here is expectancy—what to do when code falls short of expectations. Sending the graduate back to redo the cut or using the short board are both potentially viable outcomes. For example, a countertop with a gaping eighth of an inch gap between it and the wall is not going to work, but a non-load-bearing stud that's covered by drywall cut an eighth of an inch short is perfectly fine.

Conclusion

Implementation of tolerance as it pertains to peer code review is a balancing act. Like a sensitive CPU in need of a low-tolerance resistor, some pieces of functionality will require a very low code tolerance while others may not. Recognizing this will help your team foster a culture of tolerance and improve the performance of your peer review process.

Steve Bennet is a Technical Project Lead for Soliant Consulting, Inc. in Chicago IL. He's certified in some of Soliant's key areas of expertise, including PHP, Zend Framework and MySQL.

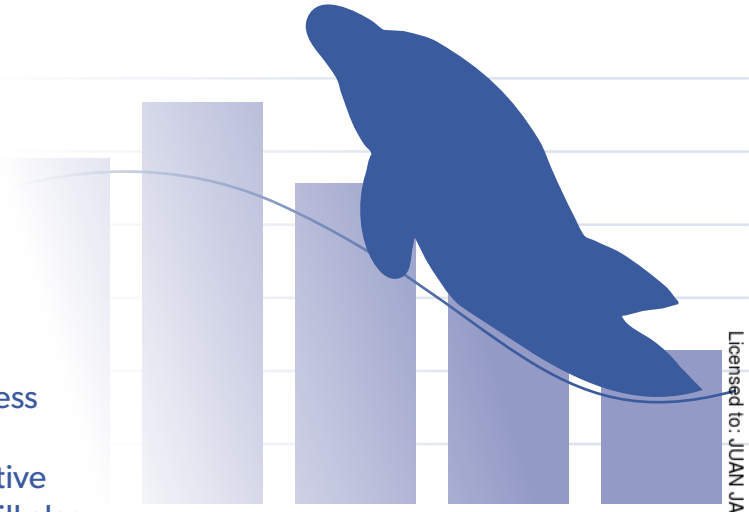
6 *Soliant:* http://www.soliantconsulting.com

7 *Development Process:* http://www.soliantconsulting.com/development-process

8 *Zend Framework:* http://www.soliantconsulting.com/web-applications/zend-solutions

Connecting Your Charts to a MySQL Database

Vikas Lalwani



This article will walk you through the step-by-step process of creating charts in PHP. First we will fetch data from a MySQL database through PHP, and then render interactive charts using that data. After making a basic chart, we will also learn about some advanced features like updating data dynamically and exporting a chart as an image. We'll make use of the FusionCharts charting library, which offers a huge collection of charts and has a dedicated plugin for PHP.

Since you are reading this right now, I can guess with almost 100 percent certainty PHP and MySQL are part of your tech stack. And if you have been doing development work for long enough then you must have dealt with making charts as well.

Then why read this article, if you already know the topic?

Well, there is a twist. Traditionally, PHP-based charting libraries render a chart as an image on the server side and send it to the client that initiated the request. There's nothing wrong with drawing charts as images, but I feel we can do better, creating interactive charts instead of static images. And that's what we are going to do in this step-by-step tutorial.

To make an interactive chart, I will first get the data from a MySQL database using PHP. Once I have the data, I will convert it into JSON format and create the chart using the FusionCharts JavaScript library[1]. I chose FusionCharts for this project as it provides a good collection of charts, is easy to learn, and works in all the browsers. Also, it has a dedicated plugin for PHP, making our job a little bit easier.

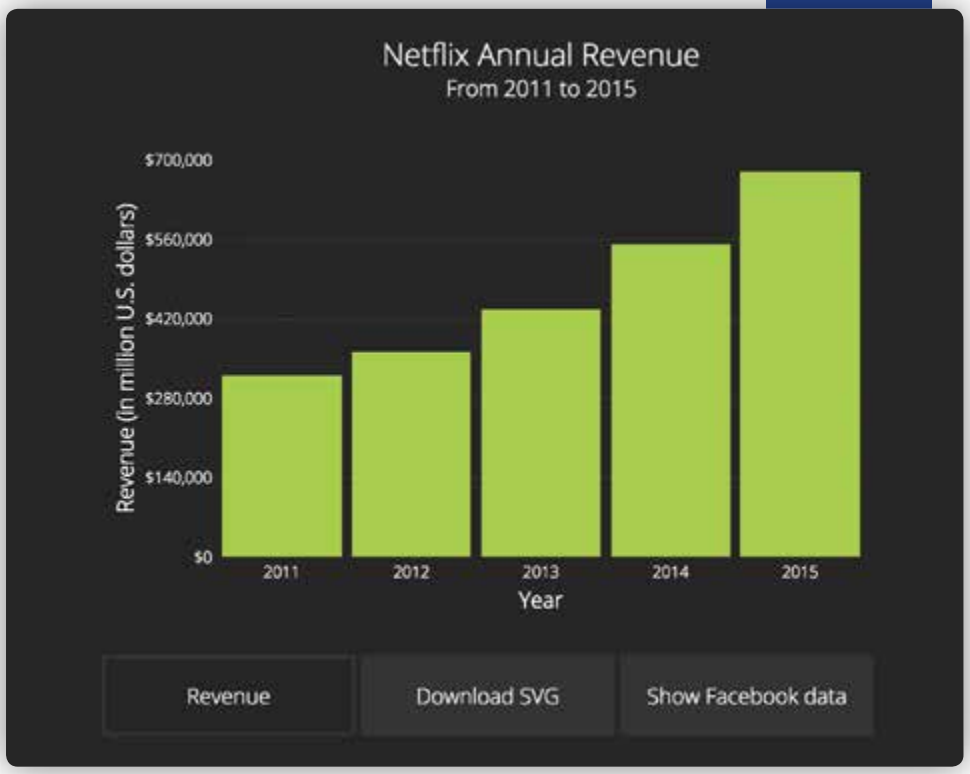
I have divided this post into two parts—in the first part I will cover the process of making a simple chart. To do that, I will query data from a database and plot a column chart, as shown in the image below. The script will be very straightforward and focus on the basics of fetching data and producing charts. Obviously, you could adapt this approach for use in your preferred framework. Once we have a basic chart ready, I will discuss some advanced charting features in the second part. That will include topics like updating the chart with new data, adding event handlers, and downloading chart data.

1 *FusionCharts:* *http://www.fusioncharts.com*

Before we get to the tutorial, here is a quick look at what our final chart will look like. You can view the source and data on GitHub, https://github.com/lalwanivikas/php-mysql-chart-demo.

It is a column chart representing the last five years' revenue for Netflix. I have made use of many chart configuration options to improve upon the chart's default styling. There is a button on the bottom right corner that you can use to update the chart with Facebook's revenue data. The button in the center will let you download the chart as an SVG file. And, finally, the box on the button's left changes its value as you hover over different columns.

Building charts using PHP and MySQL **FIGURE 1**



To get the demo running on your local system create a database and user, load the `data.sql` contents and update the database connection information. To view the charts in a browser, use PHP's built in webserver with:

```
php -S localhost:8080 -t /path/to/demo
```

Making Your First Chart: A 5-Step Process

To make it easier to understand, I have divided the complete process into the following 5 steps:

1. Include all the dependencies
2. Initiate and validate the connection to the database
3. Fetch the data through an SQL query
4. Convert the query result to a JSON array
5. Create the chart instance and close the connection to the database

Now let's explore all the steps one by one!

Step 1. Including Dependencies

For any web development project, we need to first include all the dependencies. Our current project is dependent on the following files:

- FusionCharts' JavaScript library files. You can download those from the library download page: http://www.fusioncharts.com/download/. Include both `fusioncharts.js` and `fusioncharts.charts.js` (located inside JS folder).
- FusionCharts' PHP wrapper, `fusioncharts.php`. You can download it from the plugin page: http://www.fusioncharts.com/php-charts/.

JavaScript files will go inside the HTML `<head>` using `<script>` tags.

```
<html>

<head>
  <!-- including FusionCharts JavaScript file -->
  <script type="text/javascript"
          src="/path/to/fusioncharts.js"></script>
  <script type="text/javascript"
          src="/path/to/fusioncharts.charts.js"></script>
</head>

</html>
```

The PHP wrapper will go inside PHP code as shown below.

```
// including FusionCharts PHP wrapper
include("/path/to/fusioncharts.php");
```

Step 2. Initiating and Validating the MySQL Database Connection

After including all the JavaScript and PHP dependencies, it's now time to contact the database to get the data I want to plot. Here is the PHP code to initiate connection to the database:

```
<?php

$hostdb = "localhost";  // MySQl host
$userdb = "root";  // MySQL username
$passdb = "";  // MySQL password
$namedb = "fusioncharts_phpsample";  // MySQL database name

$dbhandle = new mysqli($hostdb, $userdb, $passdb, $namedb);
```

In the above code snippet, I have included my local server values for database-related variables (`$hostdb`, `$userdb`, `$passdb`, and `$namedb`)—in your code you will have to replace those with your actual values. Remember not to commit the file with your DB credentials to your repository or use .env files.

To make sure that the connection was successful, I am going to include the code below to quickly validate it:

```
if ($dbhandle -> connect_error) {
    exit("There was an error with your connection: "
         . $dbhandle -> connect_error);
}
```

I have used `$dbhandle` to make the connection to the MySQL database in the above code. If there is any kind of error, it will return an error message. Otherwise it will proceed to execute the code discussed in the next step for fetching the data.

Step 3. Fetching Data via an SQL Query

In the previous steps, we did the initial setup, getting the files required for the project and establishing a connection to the database. In the next steps we are going to fetch the data and plot a chart using the same technique.

To get the data for Netflix's revenue, I will first form an SQL query (`$strQueryNetflix`). Usually after executing a query on a database, you will have a result set as the output; for PHP, it is commonly an array (associative or indexed).

Here is the the code to query data for Netflix's revenue:

```
// Create data for Netflix
// Form the SQL query that returns last five year's revenue
// data of Netflix
$strQueryNetflix = "SELECT Year, Revenue FROM netflix";

// Execute the query, or else return the error message.
$resultNetflix = $dbhandle->query($strQueryNetflix) or
   exit("Error: ({$dbhandle->errno}): {$dbhandle->error}");
```

`$strQueryNetflix` is the query, and the result of the query will be stored in `$resultNetflix`.

We are going to plot only Netflix's data on the initial chart load, but as described in the introduction, I will later add the functionality to update the chart with Facebook's data as well. For that purpose, I am going to keep the data ready by fetching it beforehand.

Here is the code to fetch Facebook's data:

```
// Fetch data for Facebook
// Form the SQL query that returns last five years' revenue
// data of Facebook
$strQueryFb = "SELECT Year, Revenue FROM facebook";

// Execute the query, or else return the error message.
$resultFb = $dbhandle->query($strQueryFb) or
  exit("Error: ({$dbhandle->errno}): {$dbhandle->error}");
```

With the raw data in hand, I can now move on to converting it into a format that our charting library understands.

4. Converting the Query Result into a JSON Array

As mentioned above, the output of the query will be an array. Before proceeding to making our chart using the data, I need to first convert it into a format that FusionCharts can work with (either JSON or XML).

Listing 1 is the code that appends chart configuration options along with the data, and parses the end result as a JSON array.

LISTING 1

```
01. // If the query returns a valid response, prepare the JSON
02. if ($resultNetflix) {
03.
04.   // The $arrDataNetflix array holds the chart attributes
05.   // and data
06.   $arrDataNetflix["chart"] = $commonChartAttributesArray;
07.
08.   // Create caption for NetFlix chart
09.   $arrDataNetflix["chart"]["caption"]
10.       = "Netflix Annual Revenue";
11.
12.   $arrDataNetflix["data"] = array();
13.
14.   // Push the data into the array
15.   while ($row = mysqli_fetch_array($resultNetflix)) {
16.     array_push($arrDataNetflix["data"], array(
17.       "label" => $row["Year"],
18.       "value" => $row["Revenue"]
19.     ));
20.   }
21.   // JSON Encode the data for NetFlix to retrieve the string
22.   // with the JSON representation of the data in the array.
23.   $jsonEncodedDataNetflix = json_encode($arrDataNetflix);
24. }
```

In the listing above, I am first checking to determine whether the result is valid using an if condition. If it is, then I move on to creating an associative array containing chart configuration options and the data fetched from the database. Finally, I encode the data using `json_encode` to retrieve the string containing the JSON representation of the data in the array.

> **Note:** *The variable* `$jsonEncodedDataNetflix`*, within the FusionCharts-PHP server-side wrapper, holds all the JSON data for the chart, and will be passed as the value for the data source parameter of the constructor in the next and final step.*

The design for both the charts that I am going to plot (Netflix's and Facebook's) will remain same. Only one thing will change—the chart caption. Instead of repeating the configuration settings twice, I have stored the common attributes in a variable `$commonChartAttributesArray`. This is what its content looks like:

```
// Create common chart attributes array
$commonChartAttributesArray = array(
  "subCaption" => "From 2011 to 2015",
  "captionFontBold" => "0",
  "captionFontSize" => "25",
  "captionPadding" => "50",
  // more chart configuration options
);
```

It contains chart configuration options like caption, background color, data plot color, and display formats for numbers. To learn about customizing a column chart's look and feel, you can visit this documentation page for 2D columns charts.

> **Note:** *To keep the code readable, I have only included four attributes in the above code snippet. Please refer to the GitHub repository to see all the attributes used in the chart.*

These key attributes do the following:

- `paletteColors` is used to set the color of the data plots. If you set a single color, all the columns will be that color. If you give multiple colors, the first column will take the first value, the second column will take second value, and so on.
- `plotSpacePercent` lets you control the space between the columns. The default value for this is 20 (the value is an integer, but the unit is a percentage) and the allowable range is 0–80. A value of zero will make the columns stick to each other.
- `baseFont` defines the font family of the chart. You can use any font you like; I have used Google's Open Sans in my chart. Simply include the external font file in your HTML page and set its name to baseFont attribute in order to use it in your chart.
- `bgColor` and `canvasBgColor` together allow you to control your chart's background color. Any hex color code is acceptable as input to these attributes.

If you are making a different chart, then you can search for its configuration options on the above linked page itself.

Step 5. Creating the Chart Instance and Closing the Database Connection

After the JSON array is created, I will now make a chart object using the FusionCharts constructor. I need to pass chart type, dimensions, container id, and other details to the constructor to render the chart.

Here's the syntax of FusionCharts constructor:

```
// Syntax for the instance -
$var = new FusionCharts("type of chart",
                "unique chart id",
                "width of chart",
                "height of chart",
                "div id to render the chart",
                "type of data",
                "actual data")
```

In the above syntax:

- `type of chart` defines the type of chart we are creating, in this case, `column2d`. Every chart in the FusionCharts library has a unique alias or name. You can find the alias for the chart you want to plot inside FusionCharts' documentation.
- `Unique chart id` sets the id for the chart. You should not confuse this with the id of the `<div>` element where we want to render our chart (that comes later in the list). A chart's id is used to select a chart for applying event handlers and calling methods. I will make use of it later.
- Width and height set the dimension of the chart in pixels or percentage. A value of 100 percent for width tells the chart to occupy the full container width. This will make the chart responsive.
- `type of data` means the format of data. Since I am using JSON, I will set it to JSON. Alternatively, you can use XML.
- And last, we need to set the variable that contains the chart's data—in this case, `$jsonEncodedDataNetflix`.

For our project, this is how we create a FusionCharts instance, render a chart, and finally close the database connection:

```
// FusionCharts constructor
$columnChart = new FusionCharts(
    "column2d", "myFirstChart" , "100%", 500,
    "chart-1", "json", $jsonEncodedDataNetflix
);

// Render the chart
$columnChart->render();

// Close the database connection
$dbhandle->close();
```

If you've followed everything until now, you should have a working chart sample. If not, feel free to take some time to read the source code on GitHub and try running it on your local server.

> **Note:** *Please make sure to get the chart working before proceeding to the next section, as in the remainder of the article I am going to build on what I have done previously.*

Advanced Features: Events and Methods

After conquering the basic stuff, it is time to move on to some advanced stuff! In this section I am going to talk about updating chart data, adding event handlers, and providing users with the ability to download the chart as SVG.

Updating Chart Data

Often we want to update the chart with new data. In our case, I have displayed only Netflix's data, but what if someone wants to see another company's data?

If you have been following along closely you must have noticed that we have already done the initial legwork for this by getting data for Facebook's revenue. Now we just need to use the `setChartData` method provided by FusionCharts to update the chart with new data. Listing 2 is how we use it via JavaScript. Of course, for larger datasets, you would want to load chart data on demand with AJAX, but this works to illustrate the concept.

LISTING 2

```
01. var dataFacebook = <?php echo$jsonEncodedDataFb; ?>;
02. var dataNetflix = <?php echo; $jsonEncodedDataNetflix; ?>;
03.
04. // Change Chart Data dynamically
05. function setData() {
06.   var dataButton = document.getElementById('datachange');
07.   if (dataButton.innerHTML == "Show Facebook data") {
08.     dataButton.innerHTML = "Show Netflix data";
09.     FusionCharts('myFirstChart').setChartData(dataFacebook, "json");
10.   } else if (dataButton.innerHTML == "Show Netflix data") {
11.     dataButton.innerHTML = "Show Facebook data";
12.     FusionCharts('myFirstChart').setChartData(dataNetflix, "json");
13.   }
14. }
```

I am calling `setData()` on the click of a button. To make `setChartData` work, I specified new data (`dataFacebook`) and the format in which it will be passed (`"json"`). Once it is done, our chart data will be updated whenever somebody clicks the **Show Facebook data** button.

```
<div id="datachange" class="extra"
     onclick="setData();">Show Facebook data</div>
```

To improve the experience, I have added the functionality of reverting to Netflix's data. Instead of seeing only one chart, you can toggle between the two by clicking the same button. The button text will also update accordingly.

Adding a Handler for Mouse Events

When you hover over different columns of the chart, you will notice the change in text inside the bottom left box. I've achieved this through something called `dataPlotRollOver`. It is an event raised by the FusionCharts object whenever a user hovers over a data plot (column, line, pie, etc.). We can listen to this event through an event handler function.

```
// Make sure we have the FusionCharts object before we try
// to bind events
FusionCharts.ready(function() {
  // Binding dataPlotRollOver event to feed div w/year data
  FusionCharts('myFirstChart').addEventListener(
    "dataPlotRollOver", function(ev, props) {
        document.getElementById('year')
                .innerHTML = props.value;
    }
  );
});
```

Event handler receives two arguments—`ev` and `props`. `ev` contains details related to the event—type of event, event id, etc.—while `props` contains details about the particular data plot on which the event occurred. In our case it will have the label, value, etc. of the column that we mouse over.

We are extracting the value of the column on which our mouse is present using `props.value`. You can replace `dataPlotRollOver` with `dataPlotClick` to make the event trigger on mouse click instead of mouse over.

Exporting to SVG

To enable exporting, I am going to use the `exportChart` method.

```
<script type="text/javascript">
function download_chart() {
  FusionCharts('myFirstChart').exportChart({
    "exportFormat": "svg"
  });
}
</script>

<div id="export" class="extra"
     onclick="download_chart();">Download SVG</div>
```

`exportChart` lets us export a chart as an image (SVG in our case) or as a PDF document. I just need to set the required output format and FusionCharts will do the rest. In the example above, I am triggering it by clicking the **Download SVG** button.

> **Note:** *To enable a chart to be exported, the* `exportEnabled` *attribute (of the chart object) should be set to 1. It will make use of FusionCharts' export API to export the chart. You can also export a chart at client-side by enabling the chart attribute* `exportAtClientSide`.

Next Steps

I hope this article gave you a nice overview of how to create interactive charts in your favorite tech stack—PHP and MySQL. What you learned is sufficient for making moderately complex charts, but if this area is something you would like to explore further, you may find the resources below helpful:

- **Make your charts more interactive:** In the demo above, I showed how to use `dataPlotRollOver` and `setChartData` to replicate two practical use cases. But once you start making an application, you might face a different scenario altogether. Check out events and methods offered by FusionCharts for making your charts more interactive.
- **Explore more charts:** I used a column chart in the demo, but there are more than 90 chart types that you can explore to include in your project. You know column chart doesn't work everywhere, right?
- **Making your charts look better:** Do you like playing with your chart's design to make it match the rest of your app's theme? If so, you will love playing with the hundreds of different customization options available for each chart.

Have any questions or suggestions? Feel free to catch me on Twitter: @LalwaniVikas

Vikas is a budding programmer who likes to have fun with web technologies. You can see some of his tiny experiments on his website: http://vikaslalwani.com. He is always available for a quick chat on Twitter. @LalwaniVikas

The Tangled Web We Weave

Cal Evans

This month's column is brought to you by AOL. OK, not really, but it was written by someone who works at AOL, Samantha Quiñones. In spite of the fact that she rolls her eyes at me every time I ask if she's got any of those cool AOL floppy disks (kids, floppy disks are like DVDs from the Stone Age), I asked her to share some of her thoughts on the PHP community. I know it's not the high-quality OMR you're used to in this space, but get over it. I'll be back next month yelling GET OFF MY LAWN!

We pile in the car early on a Thursday morning. Three children, two adults, four suitcases, two laptop bags, and one very large plastic dinosaur. Our mission is twofold: first, deliver the kids safely to their summer vacations; second, deliver ourselves safely to our summer vacation.

My fiancé and I are both scheduled to speak at the OpenWest conference in Sandy, Utah on July 13th. Between now and then… nothing. More than a week with no work and no responsibilities. Just the two of us in a rented Nissan Versa with a few thousand miles ahead of us.

We sleep the first night in a motel near St. Louis. As we pack up the next morning to hit the road, heading west on I-70, the thought hits. "Hey, I know someone in Kansas City. We should grab lunch!"

A few hours later we're crammed into a booth at a BBQ joint in Independence, MO, eating ribs and catching up and laughing. Sarah is a community person. A speaker and blogger, that person who answers coding questions on Slack at 1 in the morning. After lunch we get back on the road and I realize how lucky we are to have this amazing web of connections. I realize how lucky we are to wander into a strange midwestern city and have it instantly feel like home because there's been a little part of us there all along.

We turn south through Arkansas and Oklahoma. We pass through Stillwater, OK and it feels like I've been there before. I check in on Swarm and immediately friends who used to be from there are telling me where they grew up and where to go. It feels like driving through a neighborhood where you used to live, both familiar and foreign.

We drive from city to city. I start hashtagging the #GeekRoadTrip in my tweets as we share lunches and dinners and grab coffees with community people. We meet people that we've only known on Twitter and reconnect with people we can't see often enough. We gorge ourselves at a Brazilian steakhouse and spend the afternoon discussing software engineering research and our favorite TV shows. Colorado Springs becomes part of my extended home, and Albuquerque, and Flagstaff.

We visit Phoenix, where some friends have just moved. We introduce them to some community people we know there. I want them to feel at home, to be part of this.

After Phoenix we head north. We don't see as many people anymore. This part of the trip is ours to share, hiking in Sedona and at the Grand Canyon. On Monday we arrive in Salt Lake City for the conference. We start preparing for our talks and catching up on all the work we've missed.

I think back to an article I wrote a year ago while heading to another conference. I wrote about why our community is worth the effort we put into it. I had tried to capture why I feel so driven to contribute in the small ways that I can. I wrote about a feeling a sense of belonging then, and I am feeling that sense powerfully now.

I think back over the past year and the faces that define my memory of all the conferences and events. It's almost like every new thread that I add to the web is a place where I've entrusted a little part of myself. Slivers of my heart live in Madison and Salt Lake City and Seattle and Sofia and London and Louisville, kept safe for me.

It's late on Tuesday night. I procrastinated and I'm feeling stressed and panicky trying to prepare for my talk in the morning. My fiancé is sitting next to me on the bed. My fiancé, whom I met at a conference; who asked me out for the first time at a conference; who proposed to me on our way to a conference. I can't imagine how different life would be without him and without our community.

Sometimes it's hard for me to understand how warm and welcoming people can be. This community opens itself up to absolutely anyone who is willing to give even a little something back. It's remarkable. The community has given me so much and I feel like I can't come close to giving back in kind.

I think back to my very first talk, feeling taken aback by all the support I was getting from these new friends. "What did you expect?" one person asked me.

I didn't expect this.

– Samantha Quiñones

Past Events

July

php[cruise]
July 17–24, Baltimore, MD (leaving from)
https://cruise.phparch.com

LaraCon US
July 27–29, Louisville, KY
http://laracon.us

Symfony Catalunya 2016
July 22–23, Barcelona, Spain
http://symfony.cat

Upcoming Events

August

NorthEast PHP
August 4–5, Charlottetown, Prince Edward Island, Canada
http://2016.northeastphp.org

PHPConf.Asia 2016
August 22–24, Singapore
http://2016.phpconf.asia

September

SymfonyLive London 2016
September 15–16, London, U.K.
http://london2016.live.symfony.com

PNWPHP 2016
September 15–17, Seattle, WA
http://pnwphp.com/2016

DrupalCon Dublin
September 26–30, Dublin, Ireland
https://events.drupal.org/dublin2016

PHPCon Poland 2016
September 30–October 2, Rawa Mazowiecka, Poland
http://www.phpcon.pl

PHP North West 2016
September 30–October 2, Manchester, U.K.
http://conference.phpnw.org.uk/phpnw16

Madison PHP Conference 2016
September 30–October 2, Madison, WI
http://2016.madisonphpconference.com

October

LoopConf
October 5–7, Ft. Lauderdale, FL
https://loopconf.com

Bulgaria PHP 2016
October 7–9, Sofia, Bulgaria
http://bgphp.org

Brno PHP Conference 2016
October 15, Brno, Czech Republic
https://www.brnophp.cz/conference-2016

ZendCon 2016
October 18–21, Las Vegas, NV
http://www.zendcon.com

International PHP Conference 2016
October 23–27, Munich, Germany
https://phpconference.com/en/

DrupalSouth
October 27–28, Queensland, Australia
https://goldcoast2016.drupal.org.au

Forum PHP 2016
October 27–28, Beffroi de Montrouge, France
http://event.afup.org

ScotlandPHP 2016
October 29, Edinburgh, Scotland
http://conference.scotlandphp.co.uk

November

TrueNorthPHP
November 3–5, Toronto, Canada
http://truenorthphp.ca

php[world]
November 14–18, Washington D.C.
https://world.phparch.com

December

SymfonyCon Berlin 2016
December 1–3, Berlin, Germany
http://berlincon2016.symfony.com

ConFoo Vancouver 2016
December 5–7, Vancouver, Canada
https://confoo.ca/en/yvr2016

PHP Conference Brazil 2016
December 7–11, Osasco, Brazil
http://www.phpconference.com.br

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

Using Etsy/Phan for Static Analysis

Matthew Setter

Given that PHP 7 is a major version upgrade, I don't suggest you just upgrade your PHP binaries, cross your fingers, and hope for the best—not unless you're super confident your current code is PHP 7-compatible. Instead, I suggest you get a copy of an excellent tool by the engineering team at Etsy and perform some code analysis. It's called Phan, and in this month's edition, I'm going to give you a bit of an overview of it, and step you through getting it installed, finishing up by running a series of analyzes on a Zend Expressive codebase.

Unless you've had your head buried in the sand over the last 12–24 months, you'll know that PHP 7 is not only available for use but picking up ever-increasing traction within the PHP community.

If you didn't know—perhaps that's possible—depending on the report you read, PHP 7's up to twice as fast as any previous version of PHP, and uses far less memory as well.

Now, PHP is already fast. But the latest release makes it *blazingly* fast. In addition to this, it brings the following, among a host of other additions, to the language we all know and love:

- Scalar Type Hints
- Return Type Hints
- Uniform Variable Syntax

Given these changes, it's only a matter of time before it will be the standard upon which all PHP applications are built. After all, if nothing else, who wouldn't at least want a 100% performance improvement for free? I don't know about you, but I know I do.

It's an easy win. It reduces infrastructure costs. And it can make you look like a bit of a hero in the process—depending on your employer.

What is It?

According to the the GitHub repository[1], Phan is:

> *"a static analyzer that looks for common issues and will verify type compatibility on various operations when type information is available or can be deduced. Phan does not make any serious attempt to understand flow control and narrow types based on conditionals."*

Specifically, it looks for a range of potential issues with your application's source code, including the following:

- Check for all methods, functions, classes, traits, interfaces, constants, properties, and variables to be defined and accessible
- Check for type safety and arity[2] issues on method/function/closure calls
- Check for PHP 7/PHP 5 backwards compatibility
- Check for valid and type safe return values on methods, functions, and closures
- Check for unused or dead code
- Check for classes, functions, and methods being redefined

So what it will do is perform a range of checks on your codebase, regardless of its size, and print out a report showing you the issues which need addressing. What's more, you can quickly find out if your code is ready to be upgraded to PHP 7.

At this point, you may well be getting excited at the prospect of upgrading your application. But, at the same time, you may have your doubts, along with some legitimate fears, about what the analysis will show—especially if you're working on an older, legacy codebase.

Don't worry! As you'll see in a little while, Phan is quite flexible in what it looks for, and how stringent it can be.

Installation

First, though, we have to get Phan installed. Before we can, we have to ensure that the php-ast extension[3] is installed. If not, then the installation will fail, citing that it's not available. You can either install it from source[4] or you can use your package manager to install it for you.

I didn't have any luck installing from source on the Mac, but installing via Homebrew worked a treat. In addition to the `php-ast` extension, you will also need to have a PHP 7 binary available.

As with php-ast, you can likely get PHP 7 via your operating system's package manager. Alternatively, there are some excellent articles referenced in the links section.

1 *Phan Repo:* https://github.com/etsy/phan

2 *In logic, mathematics, and computer science, the arity of a function or operation is the number of arguments or operands that the function takes. The arity of a relation (or predicate) is the dimension of the domain in the corresponding Cartesian product.* https://en.wikipedia.org/wiki/Arity

3 *php-ast extension:* https://github.com/nikic/php-ast

4 *Installing Phan Dependencies:* http://phpa.me/installing-phan-deps

Assuming that both of these dependencies are available, let's get started installing Phan. Like all good PHP libraries, it's installable[5] in a number of ways, primarily by using Composer, my personal favorite.

You can also install it from source, download a Phar file, or install it using Homebrew, the Mac package manager. Check out the installation documentation[6] for further details.

To install it via Composer, from the terminal, in the root of your project run:

```
composer require --dev "etsy/phan:dev-master"
```

5 *Getting Phan Running:* http://phpa.me/getting-phan-running

6 *Phan installation documentation:* http://phpa.me/installing-phan

Running Phan

Now that it's installed, it's time to test it out. To do so, I'm going to use a Zend Expressive project. I've been working on it over the last few months, and it recently entered the final client testing phase. At 16,559 lines of code spread across 185 files, it's as good a codebase as any to use for this month's column.

We can start running a scan, directly from the command line. But Phan's documentation recommends making life easier for ourselves by creating a rather lenient configuration file. You can see it in

Copy the configuration to a file called `config.php`, located in a new directory, in the root of your project, called `.phan`. This is the default location that Phan will check for its configuration file.

The file's well documented already, but to summarize, this configuration will have Phan perform a lower intensity, quick scan, using a minimal severity level. In short, it'll go easy on your code.

LISTING 1

```
<?php
/**
 * This configuration will be read and overlaid on top of
 * the default configuration. Command line arguments will be
 * applied after this file is read.
 *
 * @see src/Phan/Config.php
 * See Config for all configurable options.
 */
return [
    // Backwards Compatibility Checking. This is slow and
    // expensive, but you should consider running it before
    // upgrading your version of PHP to a new version that
    // has backward compatibility breaks.
    'backward_compatibility_checks' => false,

    // Run a quick version of checks that takes less time at
    // the cost of not running as thorough an analysis. You
    // should consider setting this to true only when you
    // wish you had more issues to fix in your code base.
    'quick_mode' => true,

    // If enabled, check all methods that override a parent
    // method to make sure its signature is compatible with
    // the parent's. This check can add quite a bit of time
    // to the analysis.
    'analyze_signature_compatibility' => true,

    // The minimum severity level to report on. This can be
    // set to Issue::SEVERITY_LOW, Issue::SEVERITY_NORMAL or
    // Issue::SEVERITY_CRITICAL. Setting it to only
    // critical issues is a good place to start on a big
    // sloppy mature code base.
    'minimum_severity' => 10,

    // If true, missing properties will be created when they
    // are first seen. If false, we'll report an error
    // message if there is an attempt to write to a class
    // property that wasn't explicitly defined.
    'allow_missing_properties' => true,

    // Allow null to be cast as any type and for any type to
    // be cast to null. Setting this to false will cut down
    // on false positives.
    'null_casts_as_any_type' => false,

    // If enabled, scalars (int, float, bool, string, null)
    // are treated as if they can cast to each other.
    'scalar_implicit_cast' => false,

    // If true, seemingly undeclared variables in the global
    // scope will be ignored. This is useful for projects
    // with complicated cross-file globals that you have no
    // hope of fixing.
    'ignore_undeclared_variables_in_global_scope' => true,

    // Add any issue types (such as 'PhanUndeclaredMethod')
    // to this black-list to inhibit them from being reported
    'suppress_issue_types' => [
        // 'PhanUndeclaredMethod',
    ],

    // If empty, no filter against issues types will be
    // applied. If this white-list is non-empty, only issues
    // within the list will be emitted by Phan.
    'whitelist_issue_types' => [
        // 'PhanAccessMethodPrivate',
    ],

    // A list of directories that should be parsed for class
    // and method information. After excluding the
    // directories defined in
    // exclude_analysis_directory_list, the remaining files
    // will be statically analyzed for errors.
    //
    // Thus, both first-party and third-party code being used
    // by your application should be included in this list.
    'directory_list' => [
        'src',
        'vendor/',
    ],

    // A directory list that defines files that will be
    // excluded from static analysis, but whose class and
    // method information should be included.
    //
    // Generally, you'll want to include the directories for
    // third-party code (such as "vendor/") in this list.
    //
    // n.b.: If you'd like to parse but not analyze 3rd
    //       party code, directories containing that code
    //       should be added to the `directory_list` as
    //       to `excluce_analysis_directory_list`.
    "exclude_analysis_directory_list" => [
        'vendor/'
    ],
];
```

This is ideal for older codebases, where you'd likely have quite a large number of issues. It won't be too concerned about attempts to cast variables to improper types, validating missing class properties, or overridden method signatures.

But there are two key directives to focus on, `directory_list` and `exclude_analysis_directory_list`. They tripped me up at first, based on my initial reading of the documentation. They're important, because when not correctly configured, you'll get a lot of false positives reported.

You specify the source files that you want to analyze in `directory_list`, along with the supporting source files, which your source files make use of. `exclude_analysis_directory_list` lists the source files that aren't to be included in the analysis.

I made use of several third party libraries, such as Zend Framework (naturally), Interop, Flysystem by the League of Extraordinary Packages, and so on. Given that, along with my application's source files, located in `src`, I needed to include the `vendor` directory, where all the third party packages are located.

However, I'm not analyzing the third party packages, so I've added them to `exclude_analysis_directory_list`. This configuration ensures that Phan can find all the files to scan, along with all the files that they make use of, without scanning unnecessary files. I hope that helps.

With the configuration file created and saved, to run the first analysis, run the following command from the terminal in the root of your project:

```
vendor/bin/phan --progress-bar
```

This results in output similar to figure one, where you see a progress bar listing progress through the various stages of the source analysis. When it's finished, you'll then see a listing of the files which had problems, such as in the output below.

```
rc/Project/Action/ManageDocumentTrait.php:249
PhanNonClassMethodCall Call to method getNamePath on
non-class type null
src/Project/Action/RetrieveSessionUserTrait.php:20
PhanUndeclaredClassMethod Call to method get from undeclared
 class \Project\Action\Segment
src/Project/Session/GetSession.php:14
PhanUndeclaredClassMethod Call to method get from undeclared
 class \Project\Session\ContainerInterface
```

Here, you can see a sample of the files that were reported to have problems. Phan believed they were attempting to call methods from undeclared classes. In total, there were 150 issues reported. Of those, 147 were `PhanUndeclaredClassMethod`, and the other three were `PhanUndeclaredExtendedClass`.

What this means is the code was either referencing or attempting to extend an undeclared class. If you're performing an analysis while you're reading this article, you may have more issues, or you may have fewer. Let's try to get a better understanding of the types of issues and errors Phan looks for.

Error Types

Phan has a range of error types[7] for which it checks. These are summarized, from the Phan documentation, in the table below.

Inspection	Definition
Access Errors	Thrown when trying to access things that you can't access
Compatibility Errors	Thrown if there is an expression that may be treated differently in PHP 7 than it was in previous major versions
Contextual Errors	Thrown when you're performing actions out of the context in which they're allowed, such as referencing self or parent when not in a class, interface, or trait
Deprecated Error	Thrown when accessing deprecated elements, as marked by the `@deprecated` comment
NoOp Errors	Thrown when you have reasonable code but it isn't doing anything.
Parameter Errors	Thrown when you're messing up your method or function parameters in some way
Redefine Errors	Thrown when more than one thing of whatever type have the same name and namespace
Static Call Errors	Thrown if you make a static call to a non static method
Type Errors	Thrown when using incorrect types or types that cannot cast to the expected types
Undefined Errors	Thrown when there are references to undefined things
Variable Errors	Thrown when there are issues with variable usage

The specific inspections under each of these categories do a good job of identifying the key issues that make a codebase either questionable in quality or prevent it from being upgraded to the latest version of PHP.

When working with Phan, there are two key things to remember. First, make sure, as I've already mentioned, that your configuration of `directory_list`, and `exclude_analysis_directory_list` are correct. If they're not, then Phan will report a host of problems which don't exist. Second, make sure that you've annotated your code[8] both sufficiently and correctly.

The majority of the issues that I saw in my first few Phan scans were based on a misconfiguration or, I'm a little ashamed to say, incorrect code annotations. Focusing on the annotations can be helpful.

If you're serious about using Phan as part of your development tool chain, then it forces you to add at least the basic documentation to your code.

Not only will this at least help other developers, especially if they use a modern IDE, such as PhpStorm, but it may end up helping you out in the future when you're maintaining the code. After I fixed up the configuration and corrected the annotations, no further errors were reported.

7 *Issue Types Caught by Phan:* http://phpa.me/phan-issue-types

8 *DocBlock syntax:* http://phpa.me/phpdoc-syntax

Further Easing Reporting Requirements

But what if Phan wasn't as forgiving of your codebase as it was of mine? What if this initial configuration was still a little too exhausting? Say, for example, given the way the code was written, it encountered a number of empty files, for which Phan would report a `PhanEmptyFile` notification.

Or say you're using a framework, such as Oxid[9], which doesn't follow standard object inheritance patterns, and will likely result in Phan reporting a `PhanParentlessClass` notification?

If you were likely to encounter either, or both, of these situations, you could blacklist them by adding them to the `suppress_issue_types` array in `.phan/config.php`, as I have here:

```
'suppress_issue_types' => [
    'PhanEmptyFile',
    'PhanParentlessClass'
],
```

On the next scan, Phan would no longer check for them. Take a look at the issue types Phan checks for. If there are others that you're not able to correct initially, then add them to the `suppress_issue_types` array, until you're in a position to do so.

Checking Only for Specific Issues

But what about if you're only interested in a specific issue type, or subset of them? What if you're only interested in looking for classes without parents, undeclared type parameters, and a non-class method call? If you're only interested in these, you can add them to the `whitelist_issue_types` array.

When one or more issue types are added to this array, then Phan will only check for those. So to look for just the issue types mentioned, I'd update the array as follows:

```
'whitelist_issue_types' => [
    'PhanParentlessClass',
    'PhanUndeclaredTypeParameter',
    'PhanNonClassMethodCall',
],
```

Backwards Compatibility Checks

Now that we've seen how to lighten the issue checking, what about strengthening it? Specifically, what about checking that our code's ready for PHP 7? Well, we could either set `backward_compatibility_checks` to true in the configuration file. Or, we could use the short command line switch `-b`, or the longer one `--backward-compatibility-checks`.

When any one of these is in effect, Phan will report any issues found with running our application with PHP 7. If you're not sure what's changed, or what you may need to check for, definitely check out the PHP 7 migration guide[10] in the PHP manual. If you're keen to find out the other available command line switches, use the standard `-h` switch.

Reporting

Now that we've seen how to configure Phan and looked at some of the command line switches, let's turn our attention to reporting styles. The reporting type that we've seen so far is the default text format. Phan comes with four other reporting options:

- **JSON:** Helpful if you're logging or analyzing the issue type violations
- **CSV:** Helpful if you're logging issue type violations to a CSV file
- **CodeClimate:** Helpful if you're using CodeClimate to validate the quality of your codebase
- **CheckStyle:** Helpful if you're using CheckStyle as part of your tool chain, as a code quality metric

To change the default output reporting mode, you can use either the short switch `-m` or the longer switch `--output-mode`, passing one of `text`, `json`, `csv`, `codeclimate`, or `checkstyle` as the format.

Reducing Analysis Time

Finally, what if you need to check a particularly large codebase as part of your continuous deployment process, but the time required is prohibitive? In that case, you can increase the number of processes, from the default of one to the maximum supported by your testing infrastructure.

In that case, you can specify the maximum number of processes to use by passing a representative integer with the `-m` or `--processes` switch. One thing to bear in mind, though, is that not all switches, such as `--dead-code-detection` are compatible with this option.

How Far Is Your Codebase Away from PHP 7?

And that brings us to the end of this month's column. I hope that you'll consider running intermittent audits of your codebase(s) using Phan to gain a measure of their quality, or even go so far as to integrate it as a part of your continuous deployment tool chain.

Whether you're keen to ensure that your applications are ready for a seamless migration to PHP 7, or to know where they need to be improved, an investment of time in learning, configuring, and running Phan will be well worth it.

If you're already using Phan, I'd love to hear your experience on the php[architect] Facebook page[11]. Alternatively, share your thoughts on Twitter, and cc @phparch.

Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, the podcast about the business of freelancing as a software developer and technical writer, and editor of Master Zend Framework, dedicated to helping you become a Zend Framework master. Find out more http://www.masterzendframework.com.

Related Links

- Installing Phan on OS X: https://akrabat.com/installing-phan-on-os-x/
- How to upgrade php 5.6 to php 7 in XAMPP on Windows: http://phpa.me/upgrade-php7-win-xampp
- How To Upgrade to PHP 7 on Ubuntu 14.04: http://phpa.me/upgrade-php7-ubuntu1404
- Upgrade to PHP 5.6 or 7 on Mac OSX: http://phpa.me/upgrade-php7-osx

9 Oxid: http://www.oxid-esales.com/en/home.html

10 the PHP 7 migration guide: http://php.net/migration70

11 php[architect] on Facebook: https://www.facebook.com/phparch/

How We Automate With Slack

David Stockton

Like it or lump it, Slack has taken over as the corporate and community communication tool of choice. User groups, companies, interest groups, families, and friends use the tool to chat, share information, memes, and gifs, coordinate meetings and more. As a tool for corporate communication, Slack has become irreplaceable for many. And with its easy-to-use APIs, it's also possible to use Slack to automate common and tedious operations, as we'll see in this issue.

What's Slack?

For those of you who haven't run into Slack yet, it's a communications platform that is similar to IRC, but with some notable improvements. First of all, there's history, so you can look back at previous conversations even if you weren't connected to the server when the conversations took place. There are, of course, workarounds and programs that can archive IRC chat, but with Slack, you don't need to worry about it. This history is limited to 10,000 messages unless your Slack is a paid account, though. Slack supports direct, person-to-person communication, channels for different topics, as well as ad hoc group chats where you can enter a private chat with a small group for any purpose. You can easily search the archives, it has excellent mobile apps, and it's easy to use and understand.

In short, it's sort of a combination of Instant Messaging with chat rooms, with a splash of what email could be used for. If that's where it ended, it wouldn't be anything special. What really makes it shine, though, are the integrations. In IRC, you can add bots that listen to what people say in the channel and react with posting responses back in the channel. Your bot can also receive or react to external events and post messages into an IRC channel. Everything an IRC bot does is essentially the same as what a very attentive person logged into the channel could do. With Slack, you can incorporate bots as well, but there is also an API and other ways to enhance what you're able to do.

Slack Integrations

Currently, I'm in 12 different Slack teams. I have three related to my company, one for my user group, one for user group leaders, one with over 1,200 people who are developers in Denver, three for other companies, one for family, one for a book, and one for a specific PHP interest group. The team I spend the most time in is the main one for the company I work for and we have quite a few integrations. Integrations allow you to enhance the capabilities of the chat in various ways. We have integrations that add calculator, dictionary, thesaurus, gif postings, lunch coordination, Twitter, Pomodoro timers, IFTTT[1], and more.

For the more serious business-y integrations, we have a Bitbucket integration which posts information about pull requests—when they are posted, comments that are added, when they are merged, and more. We have an integration that can start a Google Hangout session in a channel or between two people who need to chat. From Jenkins, we get messages when jobs start and finish along with information about whether builds were successful, how many tests were run, and so on.

From JIRA we get information about new stories and tickets, and updates to ticket statuses. All this information is posted into specific channels based on its relevance. In some cases where there are a lot of pull requests or jobs, we've created a side-channel specifically for the posts from Bitbucket, JIRA, and Jenkins.

Up until now, everything I've mentioned is available with just a few clicks. Many of the integrations require only a single click and you've added the functionality to your Slack team. However, what I find more interesting are custom integrations that can help save time and make tedious tasks more enjoyable. That's what I want to discuss here.

Custom Slack Integrations

When Slack came out, I spent some time learning the API[2] and making some fun, silly and—sometimes—useful integrations. Back then, you could trigger a message to be sent to your own script in a couple of ways. You could either tell Slack to send every message in a single channel to your script, or you could send messages in any channel(s) that started with a particular trigger word to your script. Your script could then process the message and respond with messages of its own or cause something to happen.

My first few integrations mirrored several common IRC bot plugins. I built integrations to simulate dice rolls whenever someone posted something like `slackbot: roll a d6` to a channel. I built integrations to the JIRA API so that when people mentioned JIRA tickets, the bot would post the title, owner, priority, and status in the channel so you wouldn't have to search JIRA to determine what the ticket was about. I built a "karma" integration that kept score for any person or phrase that was posted preceded by "++". It has become our way of showing appreciation or respect for someone posting a particularly insightful or funny message.

Later on, Slack added a new feature known as "slash commands." These are commands you can post in any channel or chat that start with a forward slash. When you register a slash command, Slack will post a payload which includes the full message, the user who sent it, the channel it was in, and other metadata to whatever endpoint you choose.

Webhooks

For a number of years, I've wanted a server that was externally available on the internet and had access to some of our internal servers—at the very least, our Jenkins server. This is because all the services we use will send what's commonly referred to as "webhooks" when certain interesting events happen. Bitbucket will send a payload when pull requests are created, updated, or merged, or when comments are added,

1 *IFTTT:* https://ifttt.com

2 *The Slack API:* https://api.slack.com

and more. This allowed us to make actions occurring in Bitbucket and JIRA to cause Jenkins builds to happen immediately. Until we had this server, the best I could do was set up polling, so that twenty-four hours a day, seven days a week, every minute or two our Jenkins server would ask Bitbucket if there were any new pull requests or comments across more than a hundred repositories that the Jenkins server should know about. To me, that seems incredibly inefficient. I'd much rather have these services tell me when something happens that I care about. That's significantly more efficient, and I get to know about the relevant changes in a much more timely manner.

Soon after getting this server in place, we started setting up these webhooks for several of our repositories. We used Zend Expressive to inject the incoming webhooks and output API calls into Jenkins to start jobs. Of course, this meant creating new endpoints for each type of work we wanted to do. Since we were using a middleware approach, each little bit of middleware gets to do one job and then pass off the request to the next piece, and so on. Once the middleware we needed was built, it could be reused to more quickly set up a route and configure a middleware stack for the next webhooks integration.

But still, it was not what I dreamt of, which was something like an IFTTT interface where I could drag and drop various events and filters together with the consequences of those events at will. I wanted to be able to build any sort of integration I might need, as long as I had the building blocks of the middleware required to perform each task. We're not to that point yet, but we're getting there, and it's going to be amazing. Also, I want to be clear, while many of the ideas and architecture described in this article came from my brain, and an insignificant bit of the code came from me, the vast majority of the implementation of the services I'm going to talk about were written by my brother, Dann. He's done an amazing job getting this platform to where it currently is, and in a very short amount of time. It would not be what it is without him. So when I say "we," the credit really belongs to Dann.

Our First Slash Command

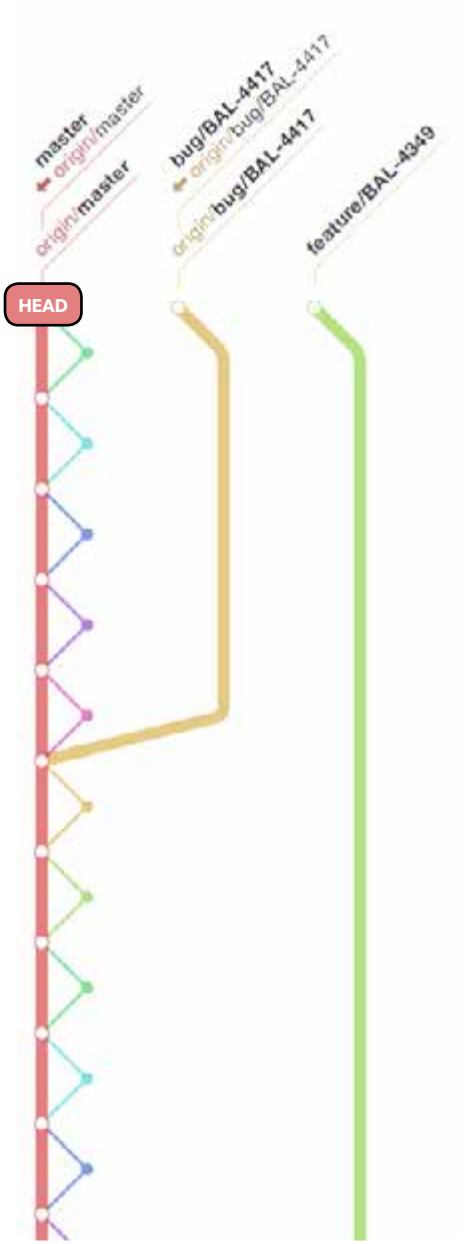
One day a few months ago, we had a team lunch. One of our remote developers needed a feature branch created by QA. Now, this team was not using git and their workflow required the feature branch to be created by QA, since developers did not have write access to the repo. I watched as our QA struggled with Bitbucket's interface on his phone, first trying to log in, then navigating to the correct repo, and finally creating a branch. It was at that lunch we decided to build a Slack slash command that would create a branch. As long as they had the Slack client, creating a branch could be as simple as writing `/branch <branch name> <repo>`. That command was finished later that day. Now QA was able to create branches easily from anywhere. We could even allow developers to create their own branches, even though they technically were not allowed to write code to this "golden" repository.

What's Next

Our other teams recently transitioned from Mercurial to git, and along with the version control changes, there were some workflow changes we could integrate because git made them possible. For the most part, the teams using git follow a rebase and squash workflow. When the code is ready to be merged, it should be a single commit branch off of the tip of the target branch. This means if several developers are working at the same time, they could each have pull requests branched off the same point in the source repository. When either of these is merged, the other branch is no longer on the tip of the target branch. We like to make each branch "zero behind" and "one ahead." Merging in this way means our source control history has a very clean "saw-tooth" style with each "tooth" being a single feature or bug fix, and the valleys between the teeth being the merges.

On all of our teams, the QA group was in charge of merging pull requests so they could control their workload and ensure that they've tested what they need to before deployment. In order to keep this saw-tooth pattern, it was necessary for QA to check the pull requests and ensure that they had the right number of approvals and nothing indicating that the code should not be merged. Then they had to change from the pull request screen to the branches screen and make sure the pull request was zero behind and one ahead. In other words, there was a lot of tedious checking needed to make sure that code was merged in the "sawtooth" and not the "foxtrot" pattern (shown in Figure1). It was an error prone process, as well, so we decided to automate it. We added onto our webhooks platform to do automatic merges when the right people have approved, the build is passing, and the branch is "0 1". Now, whenever a pull request is approved

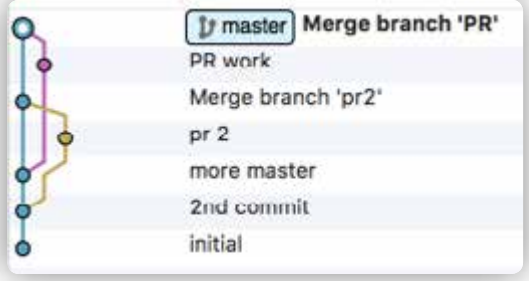
Saw-tooth pattern **FIGURE 1**



or updated, or a comment is deleted, the computer will check these criteria and merge automatically.

Sometimes, a pull request has all the approvals it needs and the builds are passing, but it's behind because something else merged in front of it—or it wasn't rebased

Foxtrot pattern **FIGURE 2**



to start when the pull request was created. Using the Slack APIs, we could notify the original developer that they needed to rebase their code. We decided to go one step further. So we built a Slack `/rebase` command to allow a Slack chat message to cause a rebase. We sent this command to the developer along with some information about what was happening. This allowed the developer to copy/paste the command back into Slack and our server would rebase on the developer's behalf, triggering a series of builds and checks, and ultimately merging the code if all the checks worked out.

Recently, Slack added a new bit of functionality allowing integrations to send messages with buttons. These buttons can do whatever you want them to. So, as of this afternoon, instead of having to copy/paste a slash command, developers can simply click a button and the server will rebase on their behalf. They don't need to stop and stash to rebase or context switch at all. Clicking the button makes it extremely simple, and making it simple means it gets done.

More Integrations

Additionally, we've built integrations allowing us to start Jenkins jobs from a Slack slash command. Since our deployments are controlled by a Jenkins job, it means certain people can deploy code from anywhere they have the internet without the need to connect to a VPN. We've built a JIRA slash command that does story lookups, but now through middleware and good code instead of the horrible, no good, very bad code of my original integration with JIRA. As more of the application was built, we decided that many of the new integrations were simply configuration. Automatically merging a repo is simply a matter of knowing what approval rules you want, potentially along with the source and destination branches. The middleware stack is the same as every other automatic merge. The same sort of thing goes for each Jenkins job that is kicked off automatically due to an event like a pull request creation, a comment added to the effect of "test this please" or "WAI U NO WORK?!?", or a merge happening. This led to the creation of two more slash commands that are, in effect, slash commands to create new integrations. We now have an `/auto_merge` and an `/auto_jenkins` command that build configuration for our middleware stacks to automatically merge or automatically kick off Jenkins jobs. These can now be done completely in Slack, requiring no code deploys or manual configuration updates. At this point, the application is entirely an API. There is no UI (yet) for anything. Everything is done through Slack or triggered by an application webhook.

This means that under normal circumstances, a developer can write their code, push up a pull request, and move on to their next task. They will be notified if anything needs to be done later, such as rebasing, or fixing things if tests failed, but they don't need to remember to go back and look at the code they completed. If they need to rebase, they can click a button. No one is held up waiting for it; there's essentially no context switching to rebase code, even if the developer is working on a different feature. Additionally, Jenkins jobs and deploys can be kicked off with a simple message from anywhere we have access to Slack.

Queuing

Fairly early on, we found that we could not have our application do the work as a direct response to the incoming request. The reason for this is that typically, webhooks require a quick, or relatively quick, response or they assume an error happened. For Slack, your application must respond in under 3 seconds. For Bitbucket, it has 10 seconds. The reason for this is that those services are sending out thousands of requests to their customers at any given time. They don't want to have their servers bogged down while an endpoint doesn't respond.

Since several operations may take a while, we found Queuing and "offline" asynchronous processing were necessary in order to respond quickly. This means for any given request, we do some minimal validation to ensure the request is legitimate and contains the information necessary to perform the action; then the request is shipped off as a message to a RabbitMQ server. On the other end of the queue is a long-running PHP job which picks up the request and makes a new HTTP request to the server again, but with a minor modification allowing it to bypass the queuing middleware without actually queuing the job. With incoming Slack requests, there's an endpoint we receive allowing us to send messages related to a slash command for up to 30 minutes after the initial slash command was issued. Our response back to Jenkins, JIRA, and Bitbucket is essentially responding, "Yup, that looks like it's a legit message," rather than a response indicating that all the work was done.

And Even More Better

Since Slack has an API and other services have APIs that provide useful functionality, the possibilities for integrations between Slack and other services, webhooks and Slack, or any other which way are virtually endless. We've built slash commands allowing us to explore Twilio logs, and look up information via our internal applications' own APIs. We also have a command which I think is one of the coolest, even if it's not necessarily the most practical.

We have a `/burrito` command. I'll say it again because it's awesome. We have a `/burrito` command in Slack. This command will literally cause an actual real-world burrito to be created and prepared for pick up. You can configure the burrito with your favorite ingredients and place the order without leaving the Slack interface. You can even re-order things you've already ordered in the past to save time. As far as silly level, this one ranks up there, but it works, it's awesome, and it's delicious.

A Brief Bit of Middleware

If you're not familiar with middleware, here's a super brief intro. A web request comes in, and it is transformed or acted upon little by little, through subsequent pieces of middleware, until a response is returned. That response passes back through each of the pieces of middleware it went through on the way in, until the response is sent back to the caller. A completely generic middleware could look like this pseudocode:

Of course, it's not necessary to build middleware that requires both a request and a response, but in a general sense, middleware will receive a request and ultimately return a response. Middleware is going to potentially modify the request or do something because of it, then pass it on to

LISTING 1

```
01. function PseudoMiddleware(Request $request,
02.                           Response $response, $next) {
03.     // do something with or modify request
04.
05.     // send in to next middleware
06.     $response = $next($request, $response);
07.
08.     // modify the response on the way back out
09.
10.     // return it
11.     return $response;
12. }
```

the next middleware piece. The response that's returned can be modified, or returned unchanged from the inner middleware. Because each middleware is so simple, the names will likely be helpful in understanding how some of our integrations work even without seeing the internal code. Some complication arises when you start dealing with errors and exceptions, but mostly that's not important right now. Here's an example of one of the middleware stacks we use. The extra levels of indentation indicate that a particular middleware is made up of other middleware. Here's our stack for automatically merging a pull request based on approvals and what-not.

```
* ValidatorMiddleware
  * ValidateBody
  * ValidateKey
  * ValidateEvent
  * ValidateRepo
  * ValidateDestination
* QueueRequestMiddleware
* BitbucketMiddleware
  * BitbucketApprovalCheck
  * BitbucketSpecificUserApproval
  * BitbucketCommentCheck
  * GitFetchCommand
  * GitVerifyBranch
  * GitAheadBehindCommand
  * BitbucketBuildStatusCheck
  * BitbucketMerge
  * JenkinsBuildWithParameters
```

As you can likely surmise, the `ValidatorMiddleware` is all about ensuring whatever comes in matches what we expect. The `ValidateKey` middleware ensures that the request has a secret shared key that the sender (Bitbucket) sends over. If any of these validators doesn't pass, the stack will return early and no further work will be done. The `QueueRequestMiddleware` puts the request into the RabbitMQ queue. On the other side, the whole process starts from the top, meaning the validation will run again, but it's quick. After that, there's a number of middlewares which ensure more specific data requirements are in place. The `BitbucketApprovalCheck` ensures that a pull request has the required number of approvals. The `BitbucketSpecificUserApproval` middleware ensures any specifically named people who must approve have done so. Next is a check to ensure that there aren't any comments that prevent merging on the actual pull request, such as "wait to merge," "this PR is not ready," or even "NO NO NO NO NO." If everything has passed so far, we get to the middleware that starts the merge. It runs a fetch, verifies that the branch we want to merge exists, and then checks that the branch is zero commits behind and one ahead (or if this was disabled, it passes through untouched). It checks to make sure all the Jenkins builds associated with this pull request have passed. It will then actually perform the merge and then optionally start another Jenkins build.

Each of these middleware pieces can be used to create other stacks for different purposes. It's a powerful way to build small bits of single-purpose code that can be used in a number of different ways. To me, it completely makes sense for this purpose. I'd recommend looking into middleware for building applications if you're not already familiar with it. It makes things seem a lot simpler, especially if you're used to working in a full stack framework.

LISTING 2

```
<?php

namespace App\Action\Bitbucket;

use App\Utility\Error;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

class BitbucketBuildCommentCheck
{
    /**
     * @var array
     */
    private $comments;

    public function __construct(array $comments) {
        $this->comments = $comments;
    }

    public function __invoke(ServerRequestInterface $request,
                             ResponseInterface $response,
                             callable $next) {
        $serverParams = $request->getServerParams();
        if (($serverParams['HTTP_X_EVENT_KEY'] ?? null)
            == "pullrequest:comment_created") {
            $body = $request->getParsedBody();
            $comment = strtolower(
                trim($body['comment']['content']['raw'])
            );
            if (!in_array($comment, $this->comments)) {
                $error = new Error(
                    "Not building PR just because you commented.",
                    ['log' => false]);
            }
        }

        return $next($request, $response, $error ?? null);
    }
}
```

Triggering a Build

This `BitbucketBuildCommentCheck` middleware in Listing 2 allows our pull requests to have special comments which start a Jenkins. Our typical configuration for these comments are "test this please", "test this plox", "test please", "test plox", "test now","u test now", or "wai u no work". If someone leaves a comment containing one of those phrases on a Pull Request, Bitbucket sends a webhook event to an endpoint specifically intended to start a Jenkins build. If the comment is not one of those, then the middleware pipeline is stopped because the comment is normal.

Queuing a Request

Listing 3 is the `QueueRequestService` class—a service which is called from middleware. It execute after a set of minimal validations have run. Those checks include things like ensuring the body of the request is JSON and contains certain key fields. If the request passes, this middleware will serialize the incoming request into Rabbit MQ and respond back with a "Processing…" message. This message will be returned back to Slack which will show the user who made that request a message. Plus since it will be fast to do this, we can get the message back to Slack quickly before it times out. Then a worker can process the job and do the real work and send additional messages to Slack once the work is completed.

LISTING 3

```
<?php

namespace App\Service\Rabbit;

use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\JsonResponse;

class QueueRequestService
{
    /**
     * @var AMQPStreamConnection
     */
    private $connection;

    public function __construct(
        AMQPStreamConnection $connection
    ) {
        $this->connection = $connection;
    }

    /**
     * @return JsonResponse
     */
    public function queueRequest(
        ServerRequestInterface $request
    ) {
        $channel = $this->connection->channel();
        $channel->queue_declare('process_requests', false,
                                true, false, false);

        $data = json_encode(['request' => serialize($request)]);
        $msg  = new AMQPMessage(
            $data, ['delivery_mode'
                    => AMQPMessage::DELIVERY_MODE_PERSISTENT]);

        $channel->basic_publish($msg, '', 'process_requests');

        $channel->close();
        $this->connection->close();

        // send response to client
        return new JsonResponse("Processing...");
    }
}
```

Validating Keys

`ValidateKey` in Listing 4 is another middleware component used in many routes. It is provided with a set of key/value pairs from a configuration file. When the middleware pipe passes through, this class pulls a "key" value from the request's query parameters and compares the expected key with one from the provided config based on the URI from the request. This key is a "shared secret" which we use to ensure that even if the endpoint is guessed, the key must be included or the middleware pipeline will stop here.

Conclusion: Stop Re-Doing It

If there's something you have to do regularly that is painful, tedious, boring, repetitive, or error-prone, I'd highly recommend automating it. Let the computer take care of the repetitive, boring parts. If you can integrate that automation in Slack, it provides a nice way for anyone in your Slack team to take advantage of the new commands and functionality. Stop doing everything manually, and get back cycles you can use on things the computer doesn't do well, like actually writing code. Have a great month—see you next time.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

LISTING 4

```
<?php

namespace App\Action\Deploy\Validators;

use App\Utility\Error;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

class ValidateKey
{
    private $keys;

    public function __construct(array $keys) {
        $this->keys = $keys;
    }

    public function __invoke(ServerRequestInterface $request,
                             ResponseInterface $response,
                             callable $next) {
        $params = $request->getQueryParams();
        $path = $request->getUri()->getPath();
        $keys = $this->keys[$path] ?? [];
        if (!in_array(($params['key'] ?? null), $keys)) {
            $error = new Error("Invalid request key.");
        }

        return $next($request, $response, $error ?? null);
    }
}
```

The Hitchhiker's Guide to Authorization

Chris Cornutt

When it comes to the security of web applications, there's a lot of things to consider. A good, secure site takes several moving pieces all cooperating in harmony to be effective. There are some pieces, however, that stand out and play a bit more prominent role. The "gateway" of any application is made up of two pieces: the authentication and the authorization. For this article, I'm going to focus on the second of this dynamic duo: authorization.

What's the Difference?

You might look at those two terms—authentication and authorization—and wonder what the difference is between them. Most developers who aren't as familiar with traditional security terms get them a bit mixed up, so here's an easy way to remember which is which:

- *authentication* answers the question "who are you?"
- *authorization* answers the question "what can you do?"

As you can see, they're basically two pieces of the same whole. In fact, in most web applications the authorization and authentication systems are intertwined into one "suite" of libraries or tools. However, they each have their own different requirements and needs. With authentication the focus is on proving the credentials provided, whatever they may be, are valid and match up with an actual user in the system. This makes it sound simple, and with a typical username/password setup it usually is, but it can get complex quickly especially if there's other systems that come into play.

Authorization is a different ballgame, though. With authentication you're basically looking for a "yes" or "no" in response to the validation. There's not a gray area where credentials are true except for one thing—either they're right or they're wrong. With authorization, this gray area definitely exists. Remember, the goal of authorization is to determine what things you have access to and can do in the system. This is going to be different from user to user and may have different answers based on different criteria.

So, enough of this overview stuff—let's get into some hard examples and get more in-depth on authorization and some of its core concepts.

None Shall Pass

As mentioned, the goal of authorization is to check and see if the current subject (user) of the system is able to perform some task or access a certain resource. This is usually done by checking one or more criteria against settings related to the current user looking for a match. Usually you're still looking for a pass/fail kind of answer here, but there's also some gray area where the user may have access to do some things and not others. For example: a user may have permission to be able to update a user record but not create a new user. They're both operations on a user instance but they're controlled by different checks.

Also keep in mind when thinking about the authorization in your application that you need to stick to the ideals of "fail fast" and the "principle of least privilege" when performing authorization validations. Basically, this means that if the criteria you're looking for don't match up, you need to immediately fail and fall back to the lowest level of functionality and permissioning available (such as "guest" for non-authenticated users or just a normal user level of access).

Types of Authorization

Let's look in more detail at the most common types of authorization you'll find in web applications. Really, the first two are the most popular of the bunch, but I want to throw in the last one (property-based) just to keep your mind open to what's out there beyond just ACLs and RBAC.

Access Control Lists (ACL)

First we'll start with one of the most common authentication types out there: the access control list. The access control list (ACL) is a pretty common feature of many different kinds of software, not just web applications. The basic idea is that you can use a list of user information as an "allowed" list (a whitelist) and if the user matches that list then they're good to go. This setup is particularly appealing early on due to its simplicity.

Listing 1 is a look at a basic example of an ACL in action.

At its most basic, that's all an ACL is—a list of criteria that's allowed past the checkpoint (our `if` statement there). If you need more than just checking against something like the username, you can step up your game and incorporate the idea of permissions into your checks. Permissions are essentially a key related back to the subject that can be checked as a yes/no flag and provide a bit more flexibility than just the one-piece check.

LISTING 1

```
01. <?php
02. $allowed = [
03.     'ccornutt',
04.     'omerida'
05. ];
06.
07. // The user has logged in and we want to see if they're in
08. // the allowed list. $user comes from a datastore somewhere
09. $username = $user->username;
10.
11. if (in_array($username, $allowed) === true) {
12.     echo 'Good to go!';
13. } else {
14.     die('Access denied');
15. }
```

LISTING 2

```
01. <?php
02. // Make a sample user instance with a "permissions" property
03. $user = new \stdClass();
04. $user->permissions = [
05.    'user-update', 'user-delete', 'user-read'
06. ];
07.
08. function has($user, $permission) {
09.    return (in_array($permission, $user->permissions));
10. }
11.
12. // Now we evaluate using our function
13. if (has($user, 'user-update') === true) {
14.    echo 'They can update the user!';
15. } else {
16.    die("Nope, can't update users!");
17. }
```

Listing 2 is an example of some code using permissions checking on a user instance to see if they can update a user.

It's not much more complex than in our first example, but you can see how to shift the checks away from just a piece of information on the user and more toward the user's related permissions. Obviously it requires a bit more engineering to get this relationship set up, but it might be worth it if you're looking for something more flexible.

The downside of access control lists like this is that they can easily become a maintenance nightmare and only provide a simple level of protection based on one piece of information. While our check above seems simple, there's a hidden danger in just checking one permission at a time. Imagine a scenario like this:

```
// Using the same $user instance as above, we need to make
// more complex checks

if ((has($user, 'user-update') && !has($user, 'user-delete')
    ) || has($user, 'user-read')) {
   echo 'Good to go...';
}
```

Now imagine needing the same kind of checks all across your codebase. That adds so much more complexity to the whole situation, and—if you're a usual reader of my column—you know that complexity is the enemy of security. All it would take is for someone to miss updating one of these `has` checks to potentially introduce a security issue. So, if we need something a bit more complex, there are other options. From here we can graduate up to "role-based access control," which gives us more grouping.

Role-Based Access Control (RBAC)

Where the normal ACL lists only check one piece of information, role-based access control (RBAC) allows for the validation of multiple values all at the same time. How is that possible, you ask? Aren't you still just checking for one match at a time? Well, the difference here is how the system is set up. In a basic ACL structure, there's a direct relationship between the user and the criteria, either a user property or a permission. With role-based access control we introduce another level that allows the grouping of permissions and relates them more to the "role" of the user.

Often this "role" will relate back to some kind of business grouping, allowing a swath of people to have the same kind of access without having to worry about adding every single permission to a user every time. You can think of roles essentially as groups. They take sets of criteria, usually permissions, and glob them together so you can easily reference the group instead of having to check every single permission inside of it.

LISTING 3

```
01. <?php
02. $user = new \stdClass();
03. $user->role = 'normal_user';
04.
05. $roles = [
06.    'normal_user' => ['user-read'],
07.    'admin_user' => [
08.       'user-read', 'user-delete', 'user-update'
09.    ]
10. ];
11.
12. function hasRole($user, $roleName) {
13.    return ($user->role == $roleName);
14. }
15.
16. function has($user, $permission, array $roles = []) {
17.    // See if it's a direct permission first
18.    if (isset($user->permissions)
19.       && in_array($permission, $user->permissions)) {
20.       return true;
21.    }
22.
23.    // Now see if it's one of the ones in our grouping
24.    $userRole = $user->role;
25.    $permissions = $roles[$userRole];
26.    if (in_array($permission, $permissions) == true) {
27.       return true;
28.    }
29.
30.    // No matching, return a failure - safety first!
31.    return false;
32. }
33.
34. if (hasRole($user, 'normal_user') === true) {
35.    echo 'Good to go!';
36.
37.    if (has($user, 'user-read', $roles) === true) {
38.       echo 'We can read users too!';
39.    }
40. }
```

Let's take our user example from earlier and update it to work with roles instead of just permissions, as in Listing 3.

It's slightly more complex than before so let's walk through it and make sure you catch what's happening. First off, the basics of it will look familiar to you from the previous ACL example. We have a check with the `hasRole` function to check the current user and see if their role matches the one you're looking for. If it does, we continue on and use another `if` to see if the user has a certain permission.

Now, the trick here is that the permissions are no longer related directly to the user's account. Instead they're linked to the group/role level. That's why when we call the `has` function on the inner `if` there's more magic happening in the function. First, it tries to look on the user itself for the `permissions` value and checks for a match. If no match is found it then starts in on the group. It locates their group (`$userRole`) and pulls in the permissions related to it for evaluation. The `has` check is then used on this grouping to see if it contains the permission we're looking for.

Sounds like a great solution, right? You can just lump people together and only have to worry about one check to see if someone has a certain role or not. It magically solves all of your problems, right? Well, like anything else, there's always a trade-off. If you're not really careful, your checks can get complex pretty quickly. For example, imagine if you had a situation like in Listing 4.

As you can see, the checks can still end up getting more and more complex. Basically, the business rule defined by the `if` check above

LISTING 4

```
<?php
$user = \stdClass();
$user->permissions = ['deny-update'];
$user->role = 'normal_user';

$roles = [
    'normal_user' => ['user-read'],
    'admin_user' => [
        'user-read', 'user-delete', 'user-update'
    ]
];

// We want to see if the user is in the admin
// group but be sure they can still update users
if ((hasRole($user, 'admin_user') === true
        && has($user, 'deny-update') === false)
    || hasRole($user, 'normal_user') === true
    && has($user, 'user-update')) {
    echo 'Good to go!';
}
```

is: if they're either in the "admin_user" role and don't have the "deny-update" permission or they're in the "normal" role and have the "user-update" permission, they're allowed to pass. We're basically back to the same issue as before, where we have these complex checks scattered around the code trying to check the same kinds of things over and over, hoping we don't miss something and make a mistake.

So, since we're all about reducing complexity, we need to figure out a different solution. There are plenty of other ways to move up from here and they'll all provide varying levels of clarity in your application, but I'd like to offer up one that many developers I've spoken with hadn't heard of before: property-based access control.

Property-Based Access Control

The basic idea behind property-based access control is the "property" in the name. In this methodology, we're not just checking for single pieces of information—we're evaluating a new kind of construct: a policy. These policies take the groupings that come with the RBAC roles and kick them up to the next level with some logic of their own. There's a standard put out by the OASIS group that's a pretty complex version of a property-based access control system called XACML[1] (XML-based). If you take the time to slog through all of the technical documentation on it you'll find that there are some interesting ideas and policies and some of the logic that can come with it.

The policies in a property-based system provide more "smarts" than just the has/has not checks of the other two options. Since there's some logic built into policies, you can get away with more things. Additionally, with a system like this, policies are generally stored in one place, making them easy to reference and reuse. This last point is important—one key to good security is reusability and consistency. By having policies you can reference that are consistent, you leave less room for error and less chance of an attacker finding a hole.

The basic idea of property-based access control is pretty simple, but there's some "bootstrapping" involved in the process. So instead, I'm going to suggest you look into a library that does some of the heavy lifting for you: psecio/propauth[2]. This library makes it simpler to set up your policies and evaluate them against a subject for a yes/no answer.

To get it installed, just use composer:

```
composer install psecio/propauth
```

Then you can start creating your first policy and running the evaluation as in Listing 5.

LISTING 5

```
<?php
require_once 'vendor/autoload.php';

use \Psecio\PropAuth\Enforcer;
use \Psecio\PropAuth\Policy;

// First we set up our user
$user = new \stdClass();
$user->permissions = ['test1'];
$user->roles = ['admin'];

// Now our policy
$policy = Policy::instance()->hasPermissions('test1')
                            ->notRoles('admin');

// And the check
$enforcer = new Enforcer();

if ($enforcer->evalaute($user, $policy) == true) {
    echo 'Good to go!';
}
```

In this basic example we're doing some of the same checking as in the ACL and RBAC examples above. We're looking at their permissions for a match and we're checking to see if they don't have an "admin" role. However, the real key here is in the reuse. No matter how many places you use that same policy object in your code, the evaluation will always be the same.

This is a pretty simple example of what the *psecio/propauth* library can do, so I highly recommend checking out its `README` and getting more information about its full list of features (including policy sets).

But Which One's Right for Me?

I've shared a lot of information with you about some common types of authorization you might see across various applications. Plenty of pros and cons come with each approach, so there's no single answer to this question. It's important to look at what sort of access control your system needs and determine which is the right fit. Sometimes you just don't need something as complex as property-based controls when a simple single-check ACL will do.

Don't assume right off the bat that the most robust and complex one is the most secure. More often than not, the security of the application comes in the application of the control, not in the control itself.

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce. @enygma

1 *XACML: https://en.wikipedia.org/wiki/XACML*

2 *psecio/propauth: http://github.com/psecio/propauth*

July Happenings

PHP Releases

PHP 7.0.7:
http://php.net/archive/2016.php#id2016-07-21-3

PHP 5.6.24:
http://php.net/archive/2016.php#id2016-07-21-4

PHP 5.5.38:
http://php.net/archive/2016.php#id2016-07-21-2

PHP 7.1.0 Beta 1:
http://php.net/archive/2016.php#id2016-07-21-1

News

Peter Petermann: Composer—What You Should Know
Peter Petermann has shared a few of his thoughts about right and wrong things to do when using Composer in your PHP-based applications. He offers suggestions based on some of the more wide-spread (but wrong, in his opinion) practices he's seen in several projects. He then breaks up the remainder of the post into various practices he's seen and calling out developers for doing including: starting a project vs installing; globally installed composer packages; tagging and building. http://phpdeveloper.org/news/24230

Joseph Silber: The new Closure::fromCallable() in PHP 7.1
In a new post to his site Joseph Silber looks at a new feature that will be coming with the next release in the PHP 7.x series - PHP 7.1 - the ability to convert a callable type into an actual Closure instance. He starts with a quick refresher on what closures/callables are in PHP (or an introduction for those not already familiar) including a simple example with the reject handling on a Laravel collection. http://phpdeveloper.org/news/24228

IBM Security Intelligence: The Webshell Game Continues
On the IBM Security Intelligence site there's a new article posted talking about webshells. For those not familiar with webshells, they're scripts that can be used to control servers or work as a platform to access other systems put in place by attackers. In this article they introduce some of the basics around webshells and the rise they're seeing in their use. http://phpdeveloper.org/news/24210

Codevate.com: Securing Client-side Public API Access with OAuth 2 and Symfony
On the Codevate.com blog there's a tutorial posted by Chris Lush showing you how to secure your client-side public API with OAuth 2 (based on the Symfony platform). He then shows how to integrate the FOSOAuthServerBundle bundle into your current Symfony-based application and the updates you'll need to make to your security.yml file. http://phpdeveloper.org/news/24201

Matt Allan: Understanding Dependency Injection Containers
In this recent post to his site Matt Allan introduces a concept that's become an integral part of most major PHP frameworks and applications recently: dependency injection containers. He then breaks down the main concept, the container, and how it is usually used to store instances of various objects and other functionality. http://phpdeveloper.org/news/24200

Liip Blog: A Quick Look on the Current State of Drupal 8 (ecosystem)
In a new post to the Liip blog Lennart Jegge shares a "quick look" at the current state of the Drupal 8 project and some of the issues some people are having making the transition. He shares some of the features he sees as still missing (a Top 10 wishlist) and how it seems difficult to get a good overview of the Drupal 8 ecosystem. http://phpdeveloper.org/news/24169

Davey Shafik: Building a Community Presence
Davey Shafik, a developer advocate at Akamai, has shared some of his own thoughts and perspective on his site about building a community presence for your company and help it "do community". He breaks it up into three man sections, filling each in with some background and concrete suggestions you can use to help get the ball rolling. http://phpdeveloper.org/news/24162

Chris Tankersley: My Docker Setup
Docker has quickly become one of the tools that have helped make difficult environment setup and thing of the past. Chris Tankersley, a PHP developer and devops practitioner, has shared some about his own PHP Docker workflow in a new post to his site. http://phpdeveloper.org/news/24247

We are in a Tech Recession

Eli White

I have never quite written a forward-looking financial article for my column, but now seems as good of time as any to do so. I am putting my flag in the ground and declaring that we are in a recession in the U.S. (if not globally), in at least the technology market.

Traditional news channels may not be forecasting financial doom yet; however, the signs are clear within the tech community, that we are beginning to feel the purse strings tighten. I know this because of discussions with various community members and companies that support Web Technology.

> *Most people consider me an optimist because I laughingly state that I would take my last two dollars and buy a money belt.*
>
> *- Zig Ziglar*

Money normally spent on training and conferences, for example, is slowing down. Often employee improvement is one of the first things to be cut to protect the bottom line since it is considered optional. Consulting rates are dropping as companies are aggressively negotiating with development shops to get more for less (or even less for less).

Even the job market is slowing down. As seen on Twitter and Facebook recently, we have many highly qualified programmers who are having trouble finding a job. Some companies are hiring, but they are overlooking those senior developers and instead looking for more affordable options.

No-one has declared a recession, but I am going to. I do not know if this is the beginning or the middle, but I am sure it is not the end. Why is it happening? Many reasons come to mind. The current election cycle in the U.S. certainly has everyone wary as to what November (and the next four years) will bring, and is certainly an active component. Perhaps it is just coinciding with a natural ebb and flow of reinvestment versus pulling internal—or perchance something deeper is at foot.

What to Do?

Knowing that this is happening, what can you do? What should you do? The best answer to that is to dig deep and prepare yourself for it. If your company is not going to reinvest in you, make sure that you do. Don't get caught on the wrong foot, if suddenly you find yourself without a position, and you do not have the resume or skill set to acquire another position quickly.

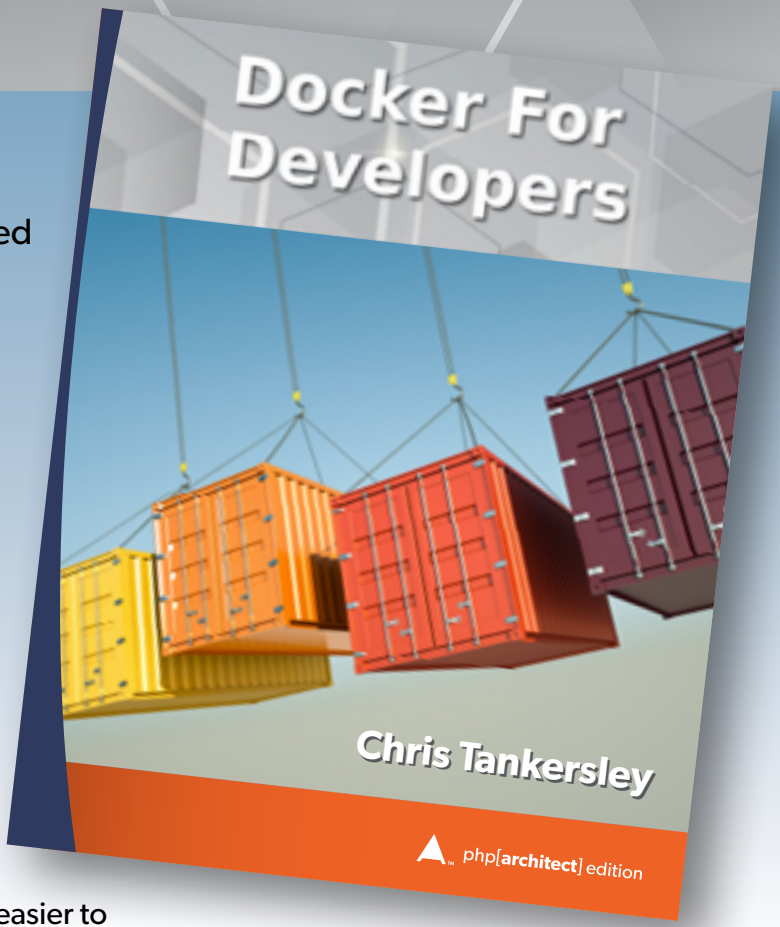
> *Remember, I'm pulling for you. We're all in this together.*
>
> *- Red Green*

Make yourself indispensable to your current company, and future companies if it comes to that. Ensure that you do not sit idle and are regularly increasing your skills and knowledge. Moreover, have some patience perhaps with the whole situation overall and attempt to ride the waves.

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. As long as he can afford some tomatoes, sausages, and nice crispy bacon, he feels he can survive any recession (or black riders). @EliW

Docker For Developers

by Chris Tankersley



Docker For Developers is designed for developers who are looking at Docker as a replacement for development environments like virtualization, or devops people who want to see how to take an existing application and integrate Docker into that workflow. This book covers not only how to work with Docker, but how to make Docker work with your application.

You will learn how to work with containers, what they are, and how they can help you as a developer.

You will learn how Docker can make it easier to build, test, and deploy distributed applications. By running Docker and separating out the different concerns of your application you will have a more robust, scalable application.

You will learn how to use Docker to deploy your application and make it a part of your deployment strategy, helping not only ensure your environments are the same but also making it easier to package and deliver.

Purchase

http://phpa.me/docker4devs

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