www.phparch.com

February 2020
Volume 19 - Issue 2

php[architect]

Cultivating the Developer Experience

Late to the Party, but Nailing It!
A Journey into Pair Programming

How to Speed up the Code Review

How We Build Sulu CMS:
10 Tips for Creating an Excellent Developer Experience

ALSO INSIDE

Free Sample Article

On Sale!
Save $200

We are the longest-running PHP developer conference in the US. This year broken into three tracks on Tech Leadership, PHP Development, and Web Technologies.

Full Schedule Published!

May 18-21, 2020

Nashville, TN — Opryland

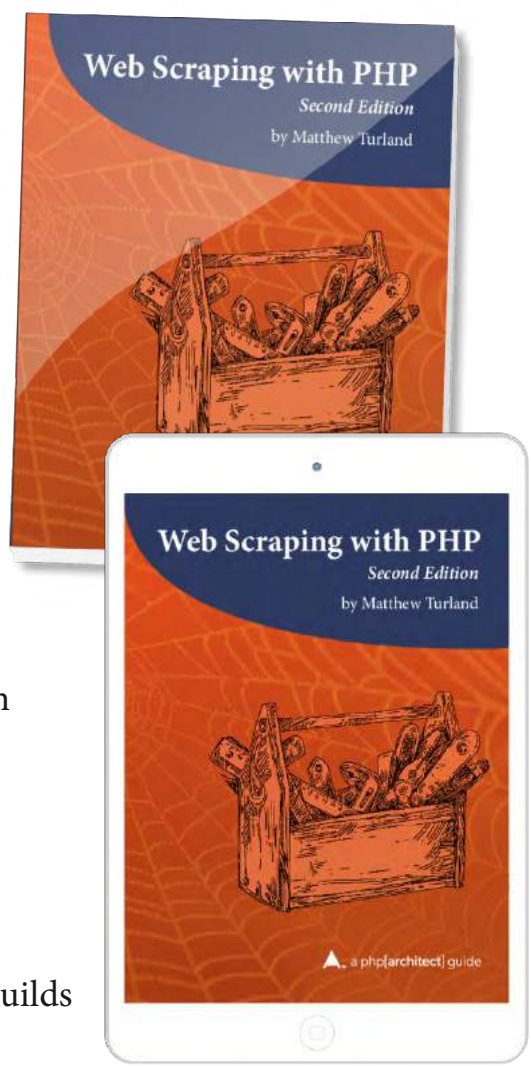
tek.phparch.com

a php[architect] guide

Read a Sample Online

Integrating with another web site but an API is not available?

Web scraping is a time-honored technique for collecting the information you need from a web page. In this book, you'll learn the various tools and libraries available in PHP to **retrieve, parse, and extract data** from HTML.



Web Scraping with PHP, 2nd Edition includes updates to the techniques of the first edition to account for modern PHP 7 based libraries written to more easily interact with web markup and data.

- HTTP requests and responses
- PHP's HTTP Stream wrapper
- Using the cURL extension
- Working with the pecl_http extension
- Parsing responses with Guzzle
- Zend Framework's HTTP classes
- An overview of Symfony's libraries for web automation
- Writing a client from scratch
- Extensions for parsing and tidying XML and HTML
- Using regular expressions
- … and more.

Written by PHP professional Matthew Turland, this book builds on his expertise in creating custom web clients.

Available in Print, PDF, EPUB, and Mobi.

Order Your Copy

http://phpa.me/web-scraping-2ed

GitHub Actions for Continuous Integration

Joe Ferguson

Continuous Integration (CI), or the ongoing process of integrating changes in a shared version control repository, should be a goal of every project you work on. This month, we're going to dive into configuring GitHub Actions to build and test our PHP application. Then we'll look at a larger scale API, which also uses GitHub Actions for Continuous Integration.

The benefit of a CI system in your software development life cycle is the automated process of running your application's build and test processes to ensure code changes won't cause unintended consequences. This automatic process also provides quick feedback to developers on their changes so they can catch bugs earlier when they're cheaper to fix instead of impacting a customer or client. You can run your CI systems in house or the cloud. GitHub You can run actions either in the cloud on GitHub's systems or in a GitHub Runner on your systems if you need to keep your builds on-premise or in your network.

Workflow

The workflow for PHP applications is generally similar to the following steps:

1. Pull the latest source code, run `composer install` to retrieve dependencies;
2. run `phpunit` or whatever your test suite of choice may be;
3. upload or share any artifacts (logs, test failure output, etc.) for later debugging;
4. and finally, notifying the user of success or failure.

There are several services and tools to accomplish this, and you may even already be using them. Travis-CI[1] is one of the most prevalent services in the open-source community because the service is free for open-source projects. CircleCI[2] is another similar service featuring a free tier and several advanced paid features such as auto-scaling and access to faster clusters. GitLab also has a service[3] and makes it easy to get up and running on your systems. You could also be using something like Jenkins[4] or TeamCity[5] to accomplish similar tasks of building and testing your application. While there is nothing wrong with these services, I was curious about GitHub's offering since I'm already using GitHub for 90% of my projects. I'm also excited by the idea I could remove the dependency on one of the several third-party applications I depend on. Even if you use a service other than GitHub, the concepts and steps required are likely similar to what I set up.

GitHub Actions allows you to run specific actions on every branch or pull request in your repository directly in GitHub. Actions are workflows you can use to build and test your projects in an automated fashion to give you more confidence in merging pull requests and fixing bugs as you can see the results of the build and test processes directly in your pull requests.

Our primary focus is to configure GitHub Actions to install our dependencies and run our tests. If everything passes, we should allow merging the pull request. We should also ensure we're testing different versions of PHP to ensure compatibility. Since this is a library which others can use, we should support as many versions as possible to allow flexibility to our potential users. My own rule of thumb is to try to support all PHP versions that are still supported by PHP.net. We can also find and use workflows created by others via the GitHub Marketplace[6] where there are currently 43 Actions matching a search query for `"PHP"`. We can start with these user-contributed workflows or build our own via the Workflow Editor. The Workflow Editor is the web-based configuration tool for GitHub Actions.

Pricing And Limits

GitHub actions are free to use with public repositories. However, if you're itching to get started with them in a private project, you need to have at least a "GitHub Pro"-level paid account. GitHub Pro includes 1GB of storage and 3,000 minutes (of actions run time) per month. "GitHub Team" level accounts increase those limits to 2GB storage and 10,000 minutes per month, while "GitHub Free" public repositories have 500MB storage and 2,000 minutes free per month.

You can choose to run your builds on Linux, Windows, or macOS. The build minutes for Windows and macOS are consumed at different rates than Linux. Windows builds consume minutes at two times while MacOS consumes build minutes at 10 times the rate of Linux systems. Given the expense can potentially get out of control, GitHub easily allows us to set and manage spending limits to ensure there are no surprises in our monthly invoice. You can find more information about setting up

1 *Travis-CI:* https://travis-ci.org

2 *CircleCI:* https://circleci.com

3 *GitLab also has a service:* https://docs.gitlab.com/ee/ci/

4 *Jenkins:* https://jenkins.io

5 *TeamCity:* https://www.jetbrains.com/teamcity/

6 *GitHub Marketplace:* https://github.com/marketplace?type=actions

these spending limits at GitHub's documentation[7].

The next limit to be aware of is the number of concurrent jobs we can have running. When starting, the GitHub Free plan limits us to 20 concurrent jobs, while the GitHub Pro limit is 40. Any jobs created while we have the maximum number of concurrent jobs already running queue automatically and execute once the queue drains.

Example Project—PHP Easy Math

We're going to ease our way into GitHub Actions by using an example repository longtime readers should recognize, PHP Easy Math[8]. We currently have the Easy Math repo configured to work with Travis-CI via the following configuration stored as `.travis.yml` in the root of our project; see Listing 1.

This configuration explains to Travis-CI our code is a PHP project which we want to execute against two different versions of PHP: **7.4** and **nightly**. We allow the **nightly** build to fail without causing the entire build to fail. Before we run our main script, we want to run `composer install`, and the main script we want to execute is `vendor/bin/phpunit`. We've created in Travis-CI the most basic PHP build setup to set up and test our project.

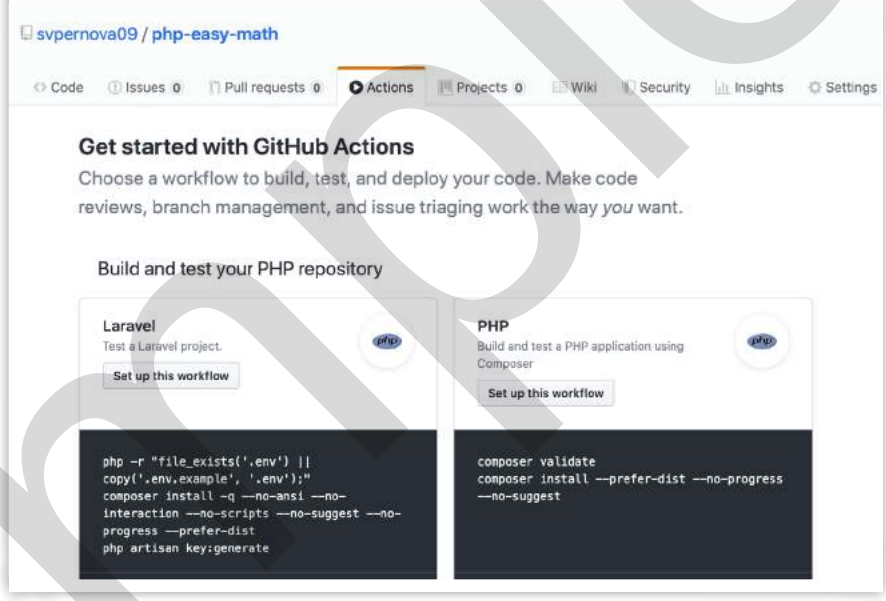
When viewing our PHP Easy Math repo, we can click on the Actions link to begin setting up our workflow (see Figure 1).

GitHub makes some assumptions based on my account and offers me Laravel and PHP workflows right off the bat. We select the option on the right to set up the generic PHP workflow and clicking on **Set up this workflow** takes us to a default configuration file. We'll proceed to commit this file to master located at `.github/workflows/php.yml` with the contents shown in Listing 2.

Listing 1

```
1. language: php
2. php:
3. - '7.4'
4. - nightly
5. matrix:
6.   allow_failures:
7.   - php: nightly
8. before_script: composer install
9. script: vendor/bin/phpunit
```

Figure 1



Listing 2

```
1. name: PHP Composer
2. on: [push]
3. jobs:
4.   build:
5.     runs-on: ubuntu-latest
6.     steps:
7.       - uses: actions/checkout@v1
8.       - name: Report PHP version
9.         run: php -v
10.      - name: Validate composer.json and composer.lock
11.        run: composer validate
12.      - name: Install dependencies
13.        run: composer install –prefer-dist –no-progress –no-suggest
14.      - name: Run test suite
15.        run: vendor/bin/phpunit
```

7 *GitHub's documentation:* https://phpa.me/github-spending-limit

8 *PHP Easy Math:* https://github.com/svpernova09/php-easy-math

> *Note that we're not using the* `--no-dev` *flag in our* `composer install` *because we want to install our testing utilities (PHPUnit). Remember to always use* `--no-dev` *in production so you don't install your testing tools, which could be an entry point for attackers*[9].

This workflow configuration file tells GitHub Actions on every push we want to run a job named `build` with the defined configuration. The build job contains the following line to run our job on the latest Ubuntu Linux image, and the `steps` section describes which actions to take.

```
runs-on: ubuntu-latest
```

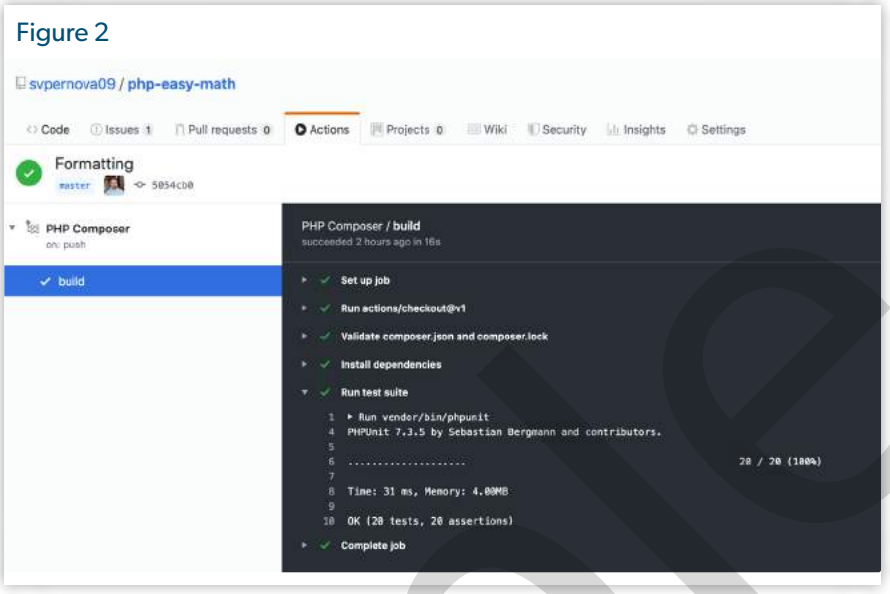
Our steps are processed in the order listed to:

1. Check out the current branch and commit,
2. report our PHP version,
3. validate our Composer files,
4. install dependencies,
5. and run our test suite.

You can see the GitHub Actions output right away via the Actions tab[10], and we can inspect the result of our Actions workflow shown in Figure 2.

We've already added some functionality we weren't previously using on Travis-CI by running `composer validate` before installing our dependencies. If you're not familiar with Composer's `validate`[11] functionality, it checks if your `composer.json` file is valid. It's designed to be used before committing any changes; however, in our use case, it catches any issues in case we forget to run validation ourselves. This line prevents us from attempting to merge changes in our Composer files, which may cause parsing errors. The validate command fails if the JSON isn't valid.

Figure 2



Listing 3

```
1. jobs:
2.   build:
3.     runs-on: ubuntu-latest
4.     strategy:
5.       fail-fast: false
6.       matrix:
7.         php: ['7.4', '7.3', '7.2', '8.0']
8.     name: PHP ${{ matrix.php }}
9.     steps:
10.      - uses: actions/checkout@v1
11.      - name: Install PHP
12.        uses: shivammathur/setup-php@master
13.        with:
14.          php-version: ${{ matrix.php }}
15.      - name: Report PHP version
16.        run: php -v
```

So far, we're only testing with the default PHP version set by GitHub Actions, which the "Report PHP version" step reports as `7.4.1`. According to *Software installed on GitHub-hosted runners*[12], other PHP versions available are 7.3, 7.2, and 7.1. Since the default seems to be 7.4, we assume we can run against any of these four versions. You may notice we're using `ubuntu-latest` which is currently the same as specifying `ubuntu-18.04`. You may want to be explicit in the image you define, as eventually the next LTS Ubuntu image becomes `ubuntu-latest`. If you need an older version, you could also specify `ubuntu-16.04` for the older LTS version.

We'll add a strategy and matrix containing an array of the PHP versions we want our project to execute against (Listing 3). We'll specify the versions of PHP we know about via our configuration except 7.1 since our Easy Math library requires at least PHP 7.2 (`.github/workflows/php.yml`).

9 *an entry point for attackers:* https://phpa.me/phpunit-install-webserver

10 *Actions tab:* https://phpa.me/php-easy-math-actions

11 `validate`: https://phpa.me/composer-validate

12 *Software installed on GitHub-hosted runners:* https://phpa.me/github-software-runners

We've added our PHP version matrix as well as named the sets of our tests the same as PHP versions, so when our action workflow runs, we get a section for each PHP version to see the results from each as in Figure 3.

Now, every time we push to a branch, GitHub Actions runs our tests and report any issues.

You might have noticed we started right off using a couple of Actions from the marketplace created by other users. The first is the `actions/checkout@v1`, which checks out our branch from version control. The second is a powerful action built by Shivam Mathur: Setup PHP[13], which is how we specify specific versions of PHP to test against. The Setup PHP Action also supports installing PHP extensions, setting INI values, and more.

The last bit of functionality to replicate from Travis-CI is to test "nightly" or the next version of PHP. Currently, this refers to PHP version 8.0. We can add this to our matrix configuration:

```
php: ['7.4', '7.3', '7.2', '8.0']
```

Since we want to allow specifically 8.0 to fail, we need to configure GitHub's "Branch protection rules" to enforce specific workflow steps to pass before a pull request can be merged. (You can also force this on administrators to keep everyone on the up-and-up). In our repository, navigate to the Settings tab and click on the Branches link in the left column. We add a rule to the `master` branch and check the option labeled "Require status checks to pass before merging," and in the Status Checks box below, we'll select PHP 7.2, PHP 7.3, and PHP 7.4 from the list. Scrolling to the bottom, we'll click Save changes to create our rule (see Figure 4).

Now, we can open a pull request, and GitHub Actions runs our workflow again. However, we can still merge this pull request even if the PHP 8.0 build fails. If we look specifically at this pull request[14], we see our required steps have passed despite the build failing on PHP 8.0 (Figure 5 on the next page). We are clear to merge to the protected master branch.

While this is not nearly as intuitive as Travis-CI's `allow_failures:` setting, it is an easy way to keep an eye on the future PHP versions you want to ensure support for. While I recommend this approach for libraries, you can remove the matrix complexity altogether and only test a specific version if your application only needs to run on PHP 7.4.

Remember, we only have a certain allotment of minutes per month, and we should put effort into caching our builds for them to run faster. By doing so, we spend fewer minutes running our builds and also get faster feedback on changes. To tell GitHub Actions

Figure 3

Figure 4

13 *Setup PHP:* https://github.com/shivammathur/setup-php

14 *this pull request:* https://phpa.me/php-easy-math-p9

to cache our dependencies, add the following two steps before our "Install Dependencies" step; see Listing 4.

When we push these changes, our Actions run checks our cache before downloading and installing the dependencies again, resulting in faster execution time. The caching is keyed to `composer.lock` file changes. Whenever you update it, the cache automatically updates on the next run.

Now, we're clear to remove the `travis-ci.yml` file from our project. We can disable the Travis-CI GitHub integration since we've duplicated the functionality in GitHub Actions.

Practical Action Usage

Example applications are all fun and games until you try to apply these ideas to a real-world application. I can share a real-world use of GitHub Actions for the Joind.in API[15], an open-source API project which powers the event speaker feedback site Joind.in. For the sake of transparency, I am one of the Joindin project maintainers, and I worked with Andreas Möller[16] who converted the Travis-CI workflow to GitHub Actions[17]. The pull request Andreas did was my first exposure with GitHub Actions, and it was a pleasant experience from the maintainer side of the project. Once we were comfortable with the workflows, it became just as invaluable as our Travis-CI configuration. Now we (Joindin) have removed that third-party process from the project, so there is one less place to have to visit to find information.

Inspecting the `.github/workflows/continuous-integration.yml` in the joindin/joindin-api repository, you can see there are multiple jobs defined where PHP Easy Math only had one job named "build." The workflow defines four jobs to execute: dependency analysis, static analysis, tests, and mutation tests. Doing so allows us to easily categorize and organize several tools to ensure not only that our tests pass, but our static analysis tools also pass. Here's an example of one way the joindin-api uses PHPStan to analyze the codebase to enforce the preferred code style:

```
- name: "Run static analysis with phpstan"
  run: vendor/bin/phpstan analyze –configuration=phpstan.neon
```

15 *Joind.in API:* https://github.com/joindin/joindin-api

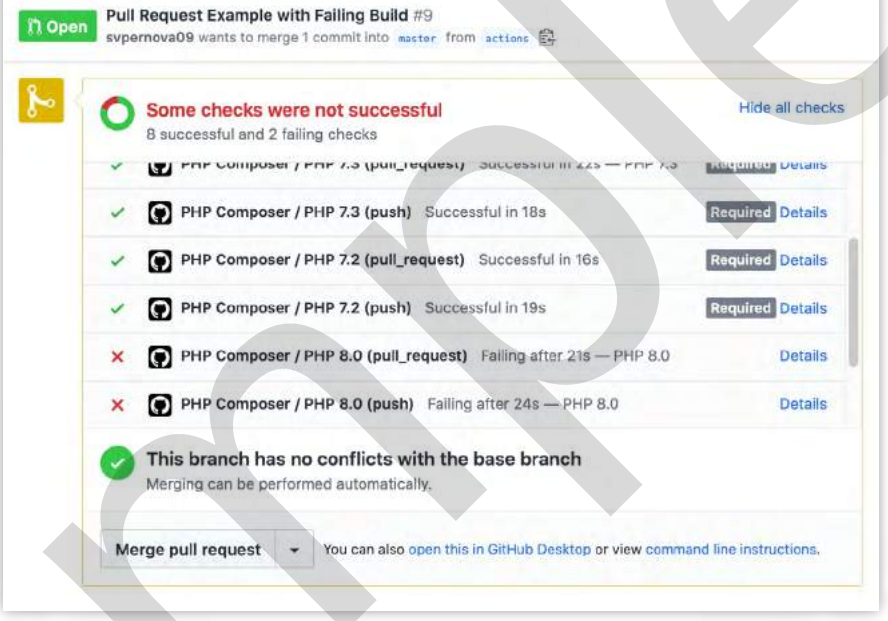
16 *Andreas Möller:* https://github.com/localheinz

17 *workflow to GitHub Actions:* https://phpa.me/joinin-api-p714

Listing 4

```
1. - name: Get Composer Cache Directory
2.   id: composer-cache
3.   run: echo "::set-output name=dir::$(composer config cache-files-dir)"
4. - name: Cache dependencies
5.   uses: actions/cache@v1
6.   with:
7.     path: ${{ steps.composer-cache.outputs.dir }}
8.     key: ${{ matrix.php }}-composer-${{ hashFiles('**/composer.lock') }}
9.     restore-keys: ${{ matrix.php }}-composer-
```

Figure 5



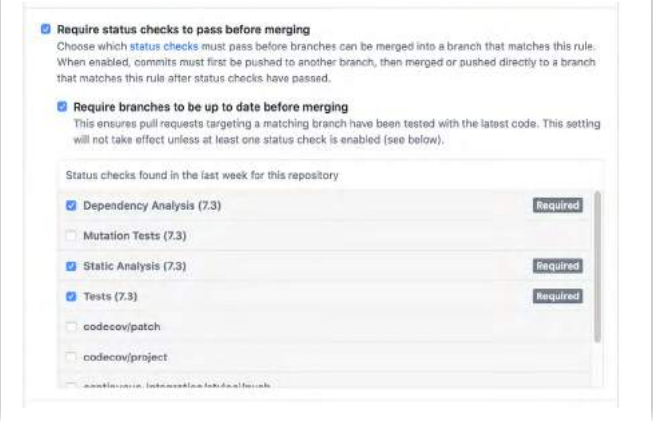
If PHPStan finds an issue, it can cause a build to fail with the error message it found, so the code quality can automatically be enforced across the entire project. I wrote about PHPStan and static analysis in the January 2019 issue[18], which you'll want to read to learn more about static analysis tools.

Much of what we've implemented in the joindin-api GitHub Actions workflow is slightly different syntax but follows the same procedure we went through with PHP Easy Math. We clone the source code, install our dependencies, check for syntax errors and code quality metrics, run our test suite, collect any errors, and upload reports to display code coverage over time. Just like our example repository, we cache dependencies, so we're able to execute our builds faster by not waiting for downloads every time.

```
- name: "Cache dependencies installed with composer"
  uses: actions/cache@v1
  with:
    path: ~/.composer/cache
    key: php-${{ matrix.php-version }}-composer-locked
-${{ hashFiles('**/composer.lock') }}
    restore-keys: |
      php-${{ matrix.php-version }}-composer-locked-
```

18 *January 2019 issue:* https://phparch.com/magazine/2019/jan/

Figure 6



If any of these steps may fail, the entire build can be failed as well. However, because there is so much happening, we may not expect everything to build perfectly all the time. If we look at the `master` branch protected rules for the joindin-api repo, we see "Dependency Analysis (7.3), Static Analysis (7.3), and Tests (7.3)" are the required jobs which must pass for the entire build to be considered passing. These settings give the flexibility of having a lot of analysis and inspection happening on the codebase while requiring the bare minimums to pass and allow deploying a build (Figure 6).

Fire Up Those Actions!

We've covered the basics and then waded to explore intermediate to advanced usage of GitHub Actions. For further reading, find their excellent documentation[19], and you'll also discover contexts and expression syntax[20] interesting. Start with the basics and don't worry if you feel overwhelmed. Your goal should be to follow at least the PHP Easy Math example of building your project by installing dependencies and then executing your test suite. Even if you don't set your builds to fail when it finds issues, this is a great start! Keep fixing the issues reported until you can enable build failures. This leads you to the Continuous Integration promise land!

Happy Actioning!

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson

Related Reading

- *Jenkins Automation* by Toni Van de Voorde, January 2019. https://phparch.com/article/jenkins-automation/
- *Making Use of Our Robot Overlords* by Brian Thompson, July 2018. https://phpa.me/robot-overlords

19 *excellent documentation:* https://phpa.me/github-actions

20 *contexts and expression syntax:* https://phpa.me/github-context-expressions

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.

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

http://phpa.me/mag_subscribe

php[**architect**]