www.phparch.com

June 2018
Volume 17 - Issue 6

php[architect]

Command and Control

Domain-Driven Architecture With Commands and Events

Pro Parsing Techniques With PHP, Part One: Simplifying Your Parsing Strategy

Self-Host Your Team's Git With Gitolite

Design Is for Designers

ALSO INSIDE

The Dev Lead Trenches: What Not To Do

Community Corner: Beyond PHP

Security Corner: Composing Application Security

Education Station: Build an API, Part Three

The Workshop: CakePHP—Part One

finally{}: Open Source & Commercial Entities

CONTENTS

php[architect]

JUNE 2018
Volume 17 - Issue 6

Features

Columns

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

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
Oscar Merida, Sandy Smith

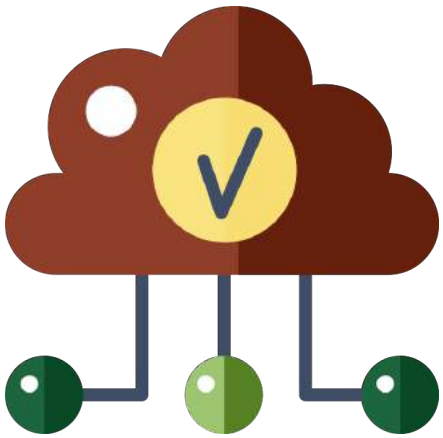
php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Self-Host Your Team's Git With Gitolite

Gabriel Zerbib



If you wish to set up a private Git server for your personal, work, or team projects, but favor free software and simple architecture, or don't want a service hosted by a third party, then Gitolite is the solution for you.

Designed in 2005 by Linus Torvalds for the needs of the Linux Kernel development team, the Git source code management system has become widely accepted outside the community. For more info check out *A Short History of Git*[1]. Free, fast, distributed, feature-rich, and yet simple to use, it has become almost indispensable today for storing, comparing, and collaborating on all types of programming projects, and even for other kinds of documents.

Options for a Private Git Repo

Although Git can be used primarily locally, its power comes from its faculties of collaborative and distributed work. A repository server is therefore central to any development project with Git as a versioning system. There are several SaaS offerings, among which is the famous GitHub, for hosting public or private Git repositories. In general, they provide services which go far beyond the versioning of the code: they range from project planning to ticket tracking, discussion forums, and even CDN for the distribution of binaries.

But some organizations prefer to move towards solutions where they retain complete control over information ownership, network topology, and administration. Free solutions[2] are not lacking to self-host a Git repository service similar to GitHub. Among the most popular tools we can list the GitLab project in Ruby, GitPrep developed in Perl, or Gogs in the Go language. Companies offer SaaS packages based on this software, but the real asset of these projects is they are free and installable on private servers.

Nevertheless, an organization that already has its tools for bugs management, project planning, and collaborative documentation may wish to simply equip itself with an internal Git server that only fulfills this function. We will present in this article the Gitolite software, which exists solely to host and control one's Git repositories on premise.

Gitolite

Gitolite[3], proposed by Sitaram Chamarty, is a collection of Perl scripts, wisely arranged to allow easy management of Git repositories served over the SSH protocol.

The tool is designed to solve one problem, and it solves it well.

Anatomy of a Git Server

Roughly, a Git repository is nothing more than a `.git` directory containing files in a particular format, which represent the entire history of changes, in all the branches. The working copy of the repository is simply a view on a given commit hash, which materializes the reconstruction of each source file as the incremental resultant of all the changes up to this level of commit. A bare clone is a clone without a materialized working copy.

By its distributed nature, when one makes a `git clone` of a repository, its entire history is fetched. Thus, each clone is itself an integral repository, and there is an equivalence of roles between the two, hence the choice of the term clone. A more in-depth study of Git's guts will bring some nuance to the above assertion, as there are more subtleties such as the configuration, reflogs, and hooks, but at first glance, we can keep this model.

Permissions support, pull requests management, tickets system, and other Wiki, although highly integrated with the GitHub platform, are satellite services outside the Git repository itself. Free products such as GitLab, which also include these modules, are therefore generally based on at least an HTTP server, a database engine, or a queue manager. The approach chosen by Gitolite is to offer control of an SSH key-based authentication layer, on top of the file system where the Git repositories are stored. The authentication is thus natively ensured by SSH, and the permissions are managed by Gitolite at repository level: the users are not Linux accounts on the server, but virtual users materialized by their public key, as we will see below.

So there is no web-based editor, no pull request tracking, or discussion thread on a commit; but there is also no database or application server! Repositories are simple folders on the server, and SSH transports the client Git protocol; it's simple, secure, and lightweight.

1 *A Short History of Git:* https://phpa.me/short-history-git

2 *Free solutions:* https://alternativeto.net/software/github/

3 *Gitolite:* https://github.com/sitaramc/gitolite

Install

At the time of writing this article, the version of Gitolite is 3.6.7. We start from a Debian Jessie or Stretch, whose `hostname` we assume is `gitserver`.

> *Note: You may also try the installation on a Docker container. You should use the* `corbinu/ssh-server` *image because Gitolite is essentially based on an SSH server, which in itself requires quite some work to assemble in Docker. Alternately, the* `jgiannuzzi/gitolite` *image is ready with Gitolite installed, and you could jump to the Configuration section.*

Preparing the Admin Workstation

Gitolite only allows key authentication, excluding any password.

To manage the server, you must have a pair of administration keys. On your station, generate it if necessary with:

```
laptop $ ssh-keygen
```

Then upload the public key to the server at a temporary location we will use later:

```
laptop $ scp ~/.ssh/id_rsa.pub gitserver:/tmp/admin.pub
laptop $ ssh gitserver chmod 444  /tmp/admin.pub
```

Store the private key carefully, as it is required to perform any administrative tasks on Gitolite, such as managing the repositories, or the user accounts. However, even if you lose it, there is a workaround explained below.

Setting Up the Server

Let's connect to the server. The installation of packages is to be executed as `root` (or with `sudo`). Let's create the `git` technical user in a non-interactive way, without any identity information thanks to the `--gecos ""` option, and with the password login method disabled.

```
gitserver# apt-get update
gitserver# apt-get install -y git-core
gitserver# adduser --disabled-password --gecos "" git
```

The Git repositories will be stored in the Home folder of the `git` user (which defaults to `/home/git`). Note that if you want the files of this technical user (and therefore all the repositories) to be hosted elsewhere, you can specify the `--home DIR` option to the `adduser` command.

Then, continuing on the server, we take the identity of `git` for the preparation of the Gitolite service.

```
gitserver# su git
gitserver$
gitserver$ cd ~
gitserver$ mkdir bin
gitserver$ echo "export PATH=$HOME/bin/:$PATH" >> .bashrc
gitserver$ git clone https://github.com/sitaramc/gitolite
gitserver$ gitolite/install -to $HOME/bin/
gitserver$ bin/gitolite setup -pk /tmp/admin.pub
```

The `gitolite/install` script prepares the executables (which are actually Perl scripts) for managing the repositories. Once in place, the `bin/gitolite` setup script prepares the file structure your meta-repository will use.

You get an output which looks like this:

```
Initialized empty Git repository in /home/git
/repositories/gitolite-admin.git/
Initialized empty Git repository in /home/git
/repositories/testing.git/
WARNING: /home/git/.ssh missing; creating a new one
    (this is normal on a brand new install)
WARNING: /home/git/.ssh/authorized_keys missing;
creating a new one
    (this is normal on a brand new install)
```

This is entirely normal, as indicated in the warning.

The installation script prepared two repositories: `gitolite-admin` and `testing`. The latter will give us the opportunity to validate our setup, which we do in the next section. The former is Gitolite's meta-repository through which the tool exposes its administration capabilities, as explained below.

Your file structure under `/home/git` (or any particular home folder you chose for this account) should now contain the following elements:

```
gitserver # ls -la /home/git

drwxr-xr-x 7 git  git  4096 Jan  7 08:19 .
drwxr-xr-x 3 root root 4096 Jan  7 08:16 ..
drwx------ 2 git  git  4096 Jan  7 08:19 .ssh
drwxr-xr-x 7 git  git  4096 Jan  7 08:19 bin
drwxr-xr-x 6 git  git  4096 Jan  7 08:19 gitolite
drwx------ 4 git  git  4096 Jan  7 08:19 repositories
```

Verification

That's all! At this stage, your `gitserver` machine is already able to serve the testing repository shipped with the installer. Let's go back to the workstation.

```
laptop $ git clone git@gitserver:testing
Cloning into 'testing'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

The workstation contacted successfully the `gitserver` server over SSH by means of your private key. The `testing` repository is empty but its successful cloning is enough to validate the efforts above.

Configuration

Once the server is up and running, it is entirely managed through the special repository `gitolite-admin`, which is owned by the administrator. That is to say, the user who owns the private key coresponding to the admin public key, as detailed earlier, is solely entitled to push changes into it. In this repository, the `master` branch is the one to which Gitolite relates to establish the currently active configuration, at any time.

The gitolite-admin Repository

Before we can create new repositories or declare users, we need to clone `gitolite-admin` on the workstation.

```
laptop $ git clone git@gitserver:gitolite-admin
Cloning into 'gitolite-admin'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
Checking connectivity... done.

laptop $ cd gitolite-admin
laptop $ tree
.
|-- conf
|   `-- gitolite.conf
`-- keydir
    `-- admin.pub
```

Listed in the `gitolite.conf` file are all the repositories of the Gitolite instance and their permissions.

The `keydir` folder contains the public keys of all the virtual users. We will look at its structure later.

Because `gitolite-admin` is in itself a cloned repository, any modification must be `git push`-ed back to the server (preceded by the appropriate `git add` and `git commit` commands) to update the effective configuration.

Repository Management

To add a repository, you simply need to declare a new section in the `conf/gitolite.conf` file in your clone of gitolite-admin, according to this format:

```
repo my-project
    RW+     =    gabriel
    R       =    alice
```

When you `push`, the Gitolite scripts automatically create the repository on the server.

To remove a repository, it is not sufficient to erase its declaration from this file, as this would simply make the repository unreachable from remote, but the corresponding folder in `home/git/repositories` would remain. You must then connect to your server with SSH and delete the folder manually:

```
gitserver $ sudo rm -rf /home/git/repositories/mon-depot
```

The configuration file follows a simple yet rich grammar [4], which allows for creating groups of users, granting permissions on specific branches using regular expressions, and fragmenting complex configurations into separate include files. For more, see the [Gitolite configuration documentation] (http://gitolite.com/gitolite/conf/)

```
@devteam1  =    gabriel chris bob
@devteam2  =    alice bob john

    repo my-project
        RW+         =    @devteam1
        R           =    jim
        RW develop  =    jim

    include "more-repos.conf"
```

User Management

A Gitolite user is an identifier declared in `gitolite.conf` as explained above. Authentication relies on the pair of keys of each user: the public keys must be stored (*commit*, *push*!) in the `keydir` folder of the `gitolite-admin` repository. At a minimum, you will find there the `admin.pub` key supplied during the install. The files must have a `.pub` extension and the file names correspond to the username for the Git repositories.

```
laptop $ cd gitolite-admin
laptop $ tree
.
|-- conf
|   `-- gitolite.conf
`-- keydir
    |-- admin.pub
    |-- alice.pub
    |-- laptop
    |   |-- admin.pub
    |   `-- gabriel.pub
    |-- desktop1
    |   `-- gabriel.pub
    `-- office
        `-- bob.pub
```

Gitolite accepts more than one public key for the same user, to allow for connecting from several machines.

Since a user corresponds to the name of a public key file, all the public keys of a given user must have the same file name below `keydir`. Fortunately, Gitolite lets you use any subfolder structure that may help you organize your keys. In the example above, we created one folder per host, in which different keys might have the same name. This choice does not prevent you from keeping key files at the top level of `keydir`. Upon incoming connection, Gitolite tries to resolve the username according to the SSH private key, by running through the `keydir` folder recursively until it finds a matching public key.

However, when connecting to the Git server with the `git` command, it is always required to use the Git SSH user:

```
git clone git@gitserver:my-project
```

The private key the underlying SSH protocol uses is indeed that of the user running the command. It is the one which Gitolite matches in order to determine the applicative account.

This mechanism is handled by the Gitolite Perl scripts which are activated when connecting to SSH via Git. This is an example of what we can find on the server in `/home/git/.ssh/authorized-keys`:

```
# gitolite start
command="/home/git/bin/gitolite-shell admin",
no-port-forwarding, no-X11-forwarding,
no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAA...MZyWKMT23X2wHbQp gabriel@laptop

command="/home/git/bin/gitolite-shell alice",
no-port-forwarding,no-X11-forwarding,
no-agent-forwarding,no-pty ssh-rsa
AAAbQ247rCFkwWx87...5CgtALOUCCIpeQ5d alice@desktop
command="/home/git/bin/gitolite-shell gabriel",
no-port-forwarding,no-X11-forwarding,
no-agent-forwarding,no-pty ssh-rsa
AAGDFGDFbo987BD...VJDF8G8BUCChvi45 gabriel@win10home
...
# gitolite end
```

The user `git` is, therefore, the one through which everything happens, thanks to SSH. Gitolite performs its tasks thanks to its own limited, custom shell. The `authorized_keys` file of use `git` is maintained by Gitolite automatically every time a *push* occurs in the *gitolite-admin* repository, as per the new updates in the `keydir` folder.

Backups

A Git repository contains its full history, so theoretically it should be enough to archive the repository's folder. However, a simple `cron` with:

```
tar czf my-project.tar.gz \
 /home/git/repositories/my-project
```

would be somewhat dangerous, and might lead to corrupted data. In practice, if the Git server is active (*push*) during the `tar` operation, the special files which represent the repository's history would be captured in an inconsistent state.

One solution could consist of temporarily stopping the SSH service on the Gitolite machine, just long enough to perform the `tar` command, to guarantee that no remote user is modifying the repository's state during the archiving. But this is not optimal and would lead to a poor user experience, especially if you manage many repositories and schedule a lot of backups.

A better approach would be to make a local clone of the repository on the server from the local path of the origin folder and then to archive this copy.

```
git clone --mirror my-project /tmp/my-project
tar czf my-project.tar.gz /tmp/my-project
```

The `--mirror` clone is also a *bare* one, which means that it does not ship a working copy. The replica has some differences with the original (in particular the hooks are lost, as mentioned above) but if the backup is mainly aiming at putting aside the source files and their history in a safe place; this method is the preferred one. The hooks are script files located directly in the `/home/git/repositories/mon-depot/hooks` folder on the Gitolite server: it is safe to archive them as regular files. Besides, an administrator should maintain them as source files on their own, with their dedicated Git repository (with scheduled backups).

Troubleshooting

Let's address some of the most frequent situations.

The Locale

If you notice all the Gcommands that you issue to your server respond with this warning:

```
laptop $ git fetch
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings
 are supported and installed on your system.

        LANGUAGE = (unset),
        LC_ALL = (unset),
        LC_PAPER = "fr_FR.UTF-8",
        LC_ADDRESS = "fr_FR.UTF-8",
        LC_MONETARY = "fr_FR.UTF-8",
        LC_NUMERIC = "fr_FR.UTF-8",
        LC_TELEPHONE = "fr_FR.UTF-8",
        LC_IDENTIFICATION = "fr_FR.UTF-8",
        LC_MEASUREMENT = "fr_FR.UTF-8",
        LC_TIME = "en_US.UTF-8",
        LC_NAME = "fr_FR.UTF-8",
        LANG = "en_US.UTF-8"

perl: warning: Falling back to the standard locale ("C").
```

Then you just need to configure the Locale environment parameters on *gitserver*. As root, add the following lines to file `/etc/environment` (or create it if needed).

```
LANGUAGE=en_US.UTF-8
LANG=en_US.UTF-8
LC_ALL=en_US.UTF-8
```

And then, still as root, execute:

```
locale-gen en_US.UTF-8
```

Locked Out?

If you ever lose your admin private key, don't panic. Everything remains possible without it (as long as you're able to connect to the server with SSH, of course).

All you have to do is log in to the server's shell, clone locally the administrative meta-repository, and declare there your new admin key.

So, first, let's generate a new key on the working station, and upload it to the server.

```
laptop$ ssh-keygen -f ~/.ssh/new-admin
laptop$ scp ~/.ssh/new-admin.pub gitserver:/tmp/
```

Now, on the server:

```
gitserver# git clone /home/git/repositories/gitolite-admin.git
gitserver# mv /tmp/new-admin.pub keydir/admin.pub
gitserver# git add keydir/admin.pub
gitserver# git commit -m "Not wise to lose the admin key"
gitserver# gitolite push
```

Notice the final Gitolite push command instead of a regular Git push! In effect, the Gitolite executable, on the server, allows bypassing the checks (hooks) during the push, which is necessary when the remote is, in fact, a local file-system path.

This operation leads to updating the `authorized_keys` file of the real `git` user, and you become able to get remote control over gitolite-admin.

Many Remotes, Many Keys

This problem is not directly related to Gitolite, but since it is very common, it is worth mentioning in this article. How can you instruct Git that it should use one particular key of yours among your very heavy keyring, when it comes to connecting to Git server?

As a matter of fact, with no particular indication, the following command:

```
git clone git@gitserver:my-project
```

will use your default private key (generally the file `~/.ssh/id_rsa`).

There are several problems here; what if the Gitolite key for my virtual user is a different file? What if I need to act as two different virtual users from the same workstation and same system account? Finally, how do I simplify the command and avoid writing the « git@ » part every time?

The solution resides in the configuration file `~/.ssh/config` on the workstation. By adding a section for our Git server, we can instruct the underlying SSH program that any `git` command uses, as to which key it should associate automatically:

```
Host gitserver
  Hostname       gitolite.domain.tld
  User           git
  IdentityFile   ~/.ssh/gitolite_gabriel_id_rsa
  IdentitiesOnly yes
```

This snippet specifies the fully qualified hostname of our Git server nickname (if needed), and which remote username should be used by default for any SSH connection (so as to avoid typing « git@ »), and the private key to communicate.

Moreover, if we add the following snippet:

```
Host git-admin
  Hostname       gitolite.domain.tld
  User           git
  IdentityFile   ~/.ssh/gitolite_admin_id_rsa
  IdentitiesOnly yes
```

you give yourself a convenient shortcut for your administrative tasks. All you have to do, is to relate to the gitserver by is new nickname *git-admin*:

```
git clone git-admin:gitolite-admin
```

It will know you want to connect with your admin user and key there.

Notice that this trick can be used with GitHub as well, if you need to work on many repositories with a unique Deploy Key[4] for each one.

Hooks

Git hooks[5] are events that the versioning system triggers upon various situations, in particular when you commit your changes.

Local Hooks

Git lets us write our own event hook handlers, in the shape of executable scripts that we put into the `.git/hooks` directory under any repository. Each hook has a fixed, pre-defined name, and if we want to activate a script every time a commit occurs, to perform some checks and accept or reject the commit, all we have to do is write an executable by the name `pre-commit` inside the hooks folder.

However, `git commit` is a local operation; your computer does not communicate with the origin repository yet. The `pre-commit` hook is a local script you can share with your friends but it does not belong to the repository—and you definitely won't persist it in the remote bank.

It is easy to install (locally) a suite of pre-commit checks for PHP projects, in order to validate the syntax, the respect of standards or to execute automatically the unit tests for example. You may find the `bruli/php-git-hooks`[6] package useful for that purpose. The local suite will do its job whether you use Gitolite or GitHub, because it only runs on your computer, without the server being aware of it.

Server-Side Hooks

There is another domain of events, which are triggered *on the server* upon network activity. The main usage is to decide whether to accept or reject a `git push`. The corresponding hook is called `update`.

In Gitolite we can configure arbitrary scripts to be wired to this hook, at the repository level, in the following steps.

1. Create a directory called `server-hooks` under `/home/git`.
2. Edit the `/home/git/.gitolite.rc` file, and just inside the `%RC` section, add the following directive.

```
%RC = (
    ...
    LOCAL_CODE   => "$ENV{HOME}/server-hooks",
);
```

4 *Deploy Key:* https://phpa.me/github-deploy-keys

5 *Git hooks:* https://phpa.me/git-hooks

6 `bruli/php-git-hooks`: https://github.com/bruli/php-git-hooks

3. Put your scripts in the `server-hooks` folder, with any name that suit your needs.
4. Then, in `gitolite-admin`, configure your various repositories in `gitolite.conf` by adding the chain of scripts you want to run at every single push command:

```
repo my-project
  RW+  = user1 user2
   ...
  - VREF/check-code-quality = @all
  - VREF/run-phpunit = @all
   ...
```

The above snippet declares the execution of the `check-code-quality` script, followed by the `run-phpunit` script in this order.

Each script, by the same name, must be an executable file under your `server-hooks` folder. They act as filters: the system calls them with the *base hash* and *target hash* as the second and third argument—and they may return a non-zero exit code to reject the push altogether.

Gitolite lets you conveniently organize your scripts atomically, and attach the relevant ones to the repositories of your choice, possibly in a different sequence for each.

In the following example, we'll see an example of a `check-code-quality` Bash script you could install to have Gitolite check the PHP files impacted by a pThe next listing is an example of a `check-code-quality` Bash script you could install to have Gitolite check the PHP files impacted by a Push operation.

Let's see in Listing 1 an example of a `check-code-quality` Bash script that you could install to have Gitolite check the PHP files impacted by a Push operation.

Conclusion

We only saw an overview of the capabilities of the tool, which offers very rich options in particular in the area of the permissions and rules at the refs level (conditions on the number of files in a push, control and authorization on specific files, working hours and more).

Besides, the program is extensible via a collection of "non-core" scripts you can activate, (for specific needs such as the support for HTTP, LDAP, replicas for high availability, or the integration with external ticket systems), which make the solution extremely flexible without compromising on security or privacy.

Gabriel has been enjoying crafting software for almost three decades, during which he learned that most technologies are pleasurable, the moment you understand which particular problem it comes to solve. Father of three, lecturer, Gabriel is currently employed at Wix.com and is always happy to share knowledge and experience. @zzgab

Listing 1

```
#!/usr/bin/env bash
BASE_HASH=${2}
PUSH_HASH=${3}

# Build the list of all the files that are in the scope
# of the tentative Push:
# - retain only the added/modified
# - consider only the files with a .php extension

TARGET_FILES=$(git diff --name-status ${BASE_HASH}..${PUSH_HASH} \
        | grep '^[ACM]' \
        | awk '{print $2}' \
        | sed -e '/\.php$/ ! d')


check_one_file() {
  local FILENAME=${1}

  # Perform here all the verifications on the changed file
  # (here we'll only run the Lint check)
  php -l ${FILENAME}
}

# From the server's point of view, the pushed files
# are in a temporary object, whose hash we know,
# and we need to materialize them in the file system
# before we can run analysis tools.
extract_and_check_file() {
  local HASH=${1}
  local FILENAME=${2}
  local TMPHASHDIR=/tmp/${HASH}
  local TMPFILE=${TMPHASHDIR}/${FILENAME}
  local check_result

  # Create a temporary folder for this specific push
  mkdir -p ${TMPHASHDIR}
  # Make Git extract the contents of the pushed file
  > ${TMPFILE} git show ${HASH}:${FILENAME}
  check_one_file ${TMPFILE}
  check_result=$?

  rm -f ${TMPFILE}
  rmdir ${TMPHASHDIR}
  return check_result
}

# Loop over all the files in the changeset,
# run our custom checks on each, exit with an
# error status (reject the Push) at first probem.
for f in ${TARGET_FILES}; do
  extract_and_check_file ${PUSH_HASH} ${f}
  is_failed=$?
  is_failed && exit is_failed
done

# No error, filter has passed
exit 0
```

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