Deploying Magento 2 in a Production Environment

Robert Egginton

13 May, 2016

In this article we'll outline an iterative deployment process for Magento 2 using git and rsync via a deployment script, and our reasons for working this way.

Magento have a help document called "Overview of deployment" which sounds promising, but it's advice on a one-off move from development to production; it says nothing about the development to production cycle. It's good to see that at least they say "Put all custom code in source control" though there's little advice as to what to keep in source control and how to update it.

A bit of background – approaches to web deployment

I'm going to go into the reasons behind why we use a build-and-sync deployment approach. If this seems obvious, then feel free to jump ahead to Deployment approach for Magento 2 or even straight to the individual commands for deployment.

0. Make changes directly on the live installation

Please don't do this.

1. Source control

Anyone who has anything more than a trivial ecommerce site knows that it's important to try out extensions and customisations on a separate development installation from your production installation.

This then begs the question of how to get these changes from development to production when these are likely to be on different machines. Source control to the rescue!

While we've all gone through a phase of putting our whole site in source control, cloning it on a live server and pointing a web-server at it, it quickly becomes apparent that any slips in source control commands or problems with composer (anyone ever had to delete the vendor folder? No? OK, you're amazing, or possibly lying) can result in a broken live site. Ecommerce site owners don't like down-time, especially unexpected down-time.

2. Build then sync

So the second phase is the use of a deployment process that does the build separately to the live site – source control, composer updates etc. – and only then copies over changed files. This approach is pretty mature and well-adopted now – Capistrano, a deployment tool in Ruby – is 10 years old this year (happy birthday, Capistrano).

Over time the best practice has become to do all possible build and compilation so that the only thing copied over is files as this is then only a single point of failure and rsync is pretty stable, giving a general approach of:

- Build/compile site off to one side
- Sync files to live
- Minimal post-sync e.g. sending an email, cleaning up permissions

Deployment approach for Magento 2

Sadly, Magento 2 does not allow us to do a clean build and sync, as certain build steps like deploying static files requires a working installation of Magento 2. This means deploying changes and then running extra steps like deploying static files, enabling modules etc.

Maintenance Mode

Apart from the issue that additional post-sync steps add points of possible failure, if we want to minimise downtime then we need to consider how long Magento 2 needs to be in maintenance mode for during a deployment.

Copying files tends to be fast enough to not require this unless you have a very busy server, and deploying static files can be included in this category. The one step that needs maintenance mode on is when enabling a module and running setup files. Magento 2 does not have the “hit the frontend and run the setup script” problems of Magento 1, but an enabled module should not be accessed without its setup scripts run.

Greaaaaat… so that are the deployment steps then?

We have ended up with the following process:

1. Check for unexpected changes
2. Update repo
3. Composer install
4. Sync files
5. Deploy static files
6. Enable maintenance mode
    1. Enable modules
    2. Run setup scripts
    3. Compile DI
    4. Clear cache
    5. Disable maintenance mode
7. Update permissions

We’ll go through these steps one at a time in a moment. First, for reference, here is the structure we use to deploy our sites:

```
/var/www/vhost/somesite/
├── deploy
│   ├── deploy.conf
│   ├── .exclude-files-deploy
│   ├── log
│   └── release-candidate
│       ├── .auth.json
│       ├── .git
│       ├── composer.json
│       ├── composer.lock
│       ├── index.php
│       ├── app
│       ├── bin
│       ├── conf
│       ├── dev
│       ├── lib
│       ├── phpserver
│       ├── pub
│       ├── setup
│       ├── update
│       ├── var
│       └── vendor
└── site
.   ├── index.php
.   ├── app
.   ├── bin
.   ├── conf
.   ├── lib
.   ├── pub
.   ├── setup
.   ├── update
.   ├── var
.   └── vendor
```

The main thing to spot here is the parallel structure in deploy/release-candidate and site. The webserver root is pointed at /var/www/vhost/somesite/site/pub/.

We have at times dispensed with the site folder and had its contents at the level above, but this adds the complication of having to exclude having to exclude itself when copying from the deploy folder to the destination live folder – this way is cleaner and allow for other files that are not part of the website repo to be stored here such as ssl certificates.

Deployment step by step

1. Check for unexpected changes

As we have a site under version control, it's a good idea to check that there have not been any changes between the last deployed version and the live code. This way we can detect any malign hacks, and also if the client has changed something without letting us know 🙂

A simple check:

```
rsync -OvrLt --delete --checksum --exclude-from='...deploy/.exclude-files-deploy' '...deploy/release-candidate/' '.../site/'
```

We can then check the output of this for "sending incremental file list" or "building file list" to see if there was a difference.

2. Update repo

Pull in the latest version of your code (or a specific release candidate commit if you work this way).

From deploy/release-candidate:

```
git fetch
git checkout origin/production
```

3. Pull down with composer

A quick reminder of the two common composer commands: `update` and `install`:

`update` is used to pull down the latest versions of required components based on restrictions in composer.json, and this updates the composer.lock file.
`install` simply takes the specific versions in composer.lock and installs them locally in vendor.

Run composer install in vendor/release-candidate to populate the vendor folder in that location rather than directly on the live site:

```
composer install --no-dev
```

4. Sync files

Rsync is the perfect tool for this task as it can use file checksums and update times to avoid copying identical files:

```
rsync -OvzrLt --delete --checksum --exclude-from='deploy/.exclude-files-deploy' 'deploy/release-candidate/' 'site/'`
```

It is useful to maintain a list of files that should not be synced. For magento2 `deploy/.exclude-files-deploy` will contain something like:

```
/app/etc/env.php
/app/etc/di.xml
/app/etc/config.php
/dev
/conf
.git
.gitignore
/var
/pub
/phpserver
/auth.json
```

Some of these are files that will not exist in the repo that should not be deleted from the site e.g. `/app/etc/env.php`. Others are folders in the repo that do not need to be deployed to a production site e.g. `/dev`.

5. Deploy static files

In theory one should only need to run the static deployment command of the Magento CLI, but we've found that this does not always update all static files that have changed, so until this is fixed it is necessary to clear the static files first. If doing this it is probably best to enable maintenance before this point:

```
rm -Rf site/pub/static
./bin/magento setup:static-content:deploy [locales]
```

where `[locales]` is the set of locales to deploy. It defaults to just `en_US` so a multi-lingual site may have a locale list of `en_GB en_US fr_FR es_ES`

Note that this command is run from the live `site/` folder. All `./bin/magento` commands from this point on are run from here.

6. Enable maintenance

This can be done simply via the magento CLI:

```
./bin/magento maintenance:enable
```

6.1 Enable modules

One of the advantages of Magento 2 is that we can choose whether to enable modules or not. One could either check in app/etc/config.php so that modules are enabled in dev and then automatically enabled by syncing this file. Alternatively, one could add an option to deployment to specify which modules are enabled. In this case, we need to run the magento CLI command to enable modules:

```
./bin/magento module:enable [module1] ... [moduleN]
```

6.2 Run setup scripts

Another easy step:

```
./bin/magento setup:upgrade
```

6.3 Compile DI

As of writing there is a current issue with compiling using the standard single-tenant compiler so you must use the multi-tenant compiler even for a single store. So the command has to be:

```
./bin/magento setup:di:compile-multi-tenant
```

As of writing there is also an issue with compilation that fails due to missing a dev-only package. Our suggested workaround is to include `"sjparkinson/static-review": "~4.1"` in the require section of `composer.json`.

6.4 Clear cache

Hurray for the CLI tool!

```
./bin/magento cache:flush
```

This clearing of all caches may be over-kill, but our experience with Magento 1 is that it is better to be safe and suffer a short performance blip.

6.5 Disable maintenance mode

Surprising nobody, it's:

```
./bin/magento maintenance:disable
```

7. Update permissions

If we're copying files as the correct user and with sensible permissions in the repo this may not be needed, but we tend to run this to be safe on production and enforce our permission schemes.

First, as we use nginx we set the group to nginx to allow read permissions for files. This might be the apache group if you use Apache. Note that we only try to change files that are in the wrong group!

```
find 'site/' ( -not -group 'nginx' ) -print -exec chgrp 'nginx' {} ;
```

The directories are set to 750:

```
find 'site/' ( -type d -not -perm '2750' ) -print -exec chmod '2750' {} ;
```

And files to 640:

```
find 'site/' ( -type f -not -perm '0640' ) -print -exec chmod '0640' {} ;
```

Done

And that's it! That's currently how we deploy Magento 2 for production. We're sure that this will evolve, and would love to hear what your thoughts are on this process.

ABOUT THE AUTHOR

Robert Egginton

As our chief problem-solver and systems architect, Rob is involved in every aspect of our development processes. Rob is partial to a bit of improvisational theatre, and setting up a smart home on a budget.

22 Comments **C3 Blog** 1 **Login**

Recommend Share Sort by Best

Join the discussion…

Михаил • a month ago
I had a problem with installation and rsync via .exclude-files-deploy.

Removing /app/etc/di.xml help me. Magento2 cannot working/setup without di.xml or something I don't understand.
• Reply • Share ›

Travis Detert • a month ago
My thoughts are that this whole process is overly cumbersome and fragile. Our experience with this nonsense thus far is that this is IMPOSSIBLE to go through without an indeterminate amount of downtime. Magento needs to address this on a fundamental level, this process is absolutely ridiculous. Adding only css changes? Deploy static content. Adding a module? di compile, and it breaks permissions often. There is no excuse for how bad this process is. They have completely undone everything that is "great" about php, just a collection of files, upload to a server, connect to database. This is ridiculous. They have taken all of the worst features of "enterprise" platforms and combined them.
• Reply • Share ›

Robert Egginton Mod → Travis Detert • a month ago
I agree that it's very complex. Part of the reason is that the Magento 2 stack (less, dependency injection, knockout JS) requires a more complex build process. However, Magento does appear to have dropped the ball in terms of considering how this would be deployed in a live environment. We have found that we can deploy the majority of our changes without downtime, but its taken ~4000 lines of deployment code to do so. And the offline part of that is still very slow. We would like to see that simplified.
• Reply • Share ›

Travis Detert → Robert Egginton • a month ago
Some good thoughts here, I agree as our frontend developers do that the knockout portion of this workflow is a huge mistake. For example: the checkout section is fundamentally ridiculous, an uncached response is currently requiring 307 http requests. Some of these are just a small snippet of javascript/json, and individual form field templates. This is absolute lunacy that they have not corrected this. If the server has any issues with network latency this creates an incredibly terrible user experience. As a php developer of nearly 18 years now, I'm absolutely shocked at how poorly this platform performs. Yesterday i was attempting to do static content deployment on about 15 themes for a multistore install, and it took over 3 hours to complete. Magento support seems to not think there's a problem here.
• Reply • Share ›

Flipmedia.co • 3 months ago
Thanks for sharing great process ;-)
• Reply • Share ›

Erfan Imani • 4 months ago
I think there's a mistake in this guide - it tells you to generate static files before you enable modules. In Magento 2.1, I think the `setup:static-content:deploy` is only available after `app/etc/config.php` exists (which comes into existence after module enabling).

This the order should be reversed - enable modules, and then deploy static files. Correct me if I'm wrong - just starting out with M2 dev/ops.
• Reply • Share ›

Robert Egginton Mod → Erfan Imani • 4 months ago
You may well be right, but I'll explain why it would not affect us: We check in config.php into the repo in order to have deployable control over enabling modules i.e. when the continuous integration server deploys the site and tests it, this will automatically include the enabling and upgrading of modules rather than leaving this as a manual step. The upshot of this is that we do not have a situation where we might want to deploy but do not have a config.php.
• Reply • Share ›

Erfan Imani → Robert Egginton • 4 months ago
Ah true, it wouldn't have an effect if it's checked in. However, it's either or. Either check the file in and don't run the enable modules command, or don't check it in and run the command (generated files shouldn't be tracked).

Reply • Share ›

Robert Egginton Mod ➦ Erfan Imani • 3 months ago

There's actually an interesting debate over where to track generated files. The downsides are obvious (repo bloat, confusion over editable files, inaccurate and messy commits), but there are upsides in certain circumstances such as the one with config.php. I've mentioned some previously, but there are more - there is an explicit history of the file that can be easily traversed, changes to enabled/disabled modules are distributed to the team so that development setups are more in sync.

A colleague recently asked me whether we could commit the core Magento files to the repo as well. At present we've not taken this step, but it would have advantages in terms of tracking what changes Magento is making, tracking transitory changes that we make when trying to fix things before moving code to separate modules etc.

So while I've always been in the "generated files shouldn't be tracked" camp myself, I've decided on the evidence that this is a little too black and white. I would now rather say "most generated files shouldn't be tracked" and then look at the particulars.

1 • Reply • Share ›

Erfan Imani ➦ Robert Egginton • 3 months ago

Awesome, all good points. Been wondering about committing core Magento files (or the vendor directory for that matter) as well..

One more question, the enable modules command does nothing more than generate config.php right? So if it's tracked, you wouldn't need to run the enable modules command?

I'll probably start tracking config.php since I can control which modules get enabled/disabled, seems like the way to go.

• Reply • Share ›

Robert Egginton Mod ➦ Erfan Imani • 3 months ago

That's right. We only run setup:upgrade on production. module:enable is only run in the dev environment, and then config,php is committed.

It's a way of working :)

1 • Reply • Share ›

chollie • 5 months ago

Just on a side note, what default Magento 2 modules do you recommend on disabling?

• Reply • Share ›

Robert Egginton Mod ➦ chollie • 5 months ago

If I suggested doing so above maybe I wasn't quite clear, as I don't think we do. I know that in Magento 1 we would disable modules like Polls if we didn't need them (of course they did not bother to port Polls to M2 due to hardly anyone using them :)). We have not yet experimented with a similar approach in M2 yet, though it should be pretty safe.

• Reply • Share ›

chollie • 5 months ago

Why do you need to rsync? Is there any reason why you don't symlink the docroot to the repo?

• Reply • Share ›

Robert Egginton Mod ➦ chollie • 5 months ago

We would not want to symlink to a single RC folder as that would negate the point of deployment. I know that some people like to deploy by creation multiple folders and then switch over the symlink. This has the additional advantage of allowing very fast roll-back. The drawbacks are that shared and transient data needs to be copied or symlinked into these in turn, which is itself a cause of error; and there is no opportunity to double-check the difference in files, though one could always use an rsync dry-run to do this. So I would say that there is no overwhelming reason to use rsync over symlinking.

• Reply • Share ›

Sam Tay • 6 months ago

I'm under the impression that `rm -rf pub/static` is actually a bad idea, because there is an important .htaccess file in pub/static and when it is missing, symlinking in dev mode breaks. However, dev mode wouldn't be run in a production environment anyway so maybe this is a non-issue. Thoughts?

• Reply • Share ›

Robert Egginton Mod ➦ Sam Tay • 6 months ago

While we do not run this script in dev mode, I do clear down pub/static in dev mode. I've not had a problem with symlinking, but we exclusively use nginx so any issues with .htaccess files would not be picked up by us.

• Reply • Share ›

Klaas van der Weij • 7 months ago

Hi Robert,

Thanks for the good read and advice. This sort of information surely lacks from the documentation provide by Magento and it needs to be out there for the community, so well done!

I'm developing a tool to perform automatic deployments for our companies setup and it resembles this structure a lot. However I enable maintenance mode earlier in the process, namely before syncing the files and deploying the static content. I suspect that in the process you describe some problems could occur when between step 4 and 6 because the new code is used/executed before upgrading the database and compiling DI. Especially because step 5 takes quite/very long.

The long process of deploying static content is quite a pain in the bun, so I also prefer to take this _outside_ of maintenance. For now I have the deployment mode set to "default", and perform step 5 _after_ leaving maintenance. This way static content is either generated by requests or picked up by the slow deployment process.

Do you think my approach is viable?

Reply • Share ›

Robert Egginton Mod → Klaas van der Weij • 7 months ago

Hi Klaas. Thanks - I'm glad it's been useful. With regards static content files we actually found that we needed to delete the current static files to be certain that they were being correctly created (not 100% certain that this is required, but we were being on the safe side). This did therefore require putting it into maintenance mode for the whole of static deployment, which made deployments far too slow in terms of down-time. We now do static deployment in RC and rsync across, though as this has its own difficulties we'll be writing another post on this.

With respect to the idea of using default mode, Magento do not recommend using this for production as it's not supposed to be as quick, and I'm not sure that css/js merging and bundling will work correctly. But I have not done any speed comparisons myself to know how important this is.

Reply • Share ›

Klaas van der Weij → Robert Egginton • 7 months ago

I have indeed disregarded Magento's recommendation to use production mode for the production environment. Speed-wise I have not done any comparisons, but we're happy with it. We do not use JS & CSS merging as this causes the admin to be tearjerkingly slow. Now you mention it, that it probably caused by the deployment mode.

Generating the static files before syncing sound like a good plan and I'll look into that aswell. I'll keep an eye open for the next post on this subject. It's great you share this knowledge because it's quite the pioneering work. Thanks for this

Reply • Share ›

Gary Mort • 8 months ago

Do step 5 on the release candidate server, not the production server.
Do step 6.3 on the release candidate server, not the production server.

Add the following to your rsync command AFTER --exclude:
--include-from='.include-files-deploy'
-------include-files-deploy-----
/pub/static/
/var/generation/
/var/di/

Sequence in the command is important. Includes that are specified after excludes overrides those excludes. Excludes specified after includes do the same.

Now you go from step 4 to step 6, 6.1, and 6.2

Repeat step 4 because Magento 2 deleted a large number of your generated files when you ran setup:upgrade - so you can pull all those generated files over from your release server. [technically you could run separate commands but rsync is so dang fast, why bother?]

Now continue with 6.3

see more

Reply • Share ›

Robert Egginton Mod → Gary Mort • 7 months ago

Thank you very much for your detailed reply! This is very much the approach that we've ended up with. Previously we did not do it this way as static deployment and DI compilation require live database access which concerned us, but does not appear to cause an issue in practice.

The greater problem was that, as you point out, running setup scripts can affect static content. Given that with a number of locales we could be looking at 20 minute to compile static files, we do not want to run these scripts on the live database and then perform sync and/or static deployment afterwards as these would be out of sync. The answer for us has been to copy the live database, run the setup scripts on that and then perform static compilation and DI compilation. This can then be rsynced across safely and quickly.

Although in theory we could skip step 7, we do not because this is there for safety in case there's been a change that we have not considered - a client has uploaded a file, or something has been run as root by accident.

It's interesting to see your take on rsync commands. As you rightly point out, it's important to know what you are going to change in case you've made a mistake or something has changed on live that you did not know about (we have picked up recent exploits in the past this way). We do not pipe to another file from rsync, but instead we run it in dry-run mode first and output the itemised result so that we can eyeball it. Rsync caches the result of dry-runs so the next run without it does not need to do the full comparison before syncing.

I wasn't aware of the --delete-after option. Thanks for that - I'll be using that to make the dry-run results easier to follow.

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

LATEST TWEETS

78 days ago
Read our latest blog post on how to programatically clear Magento 2 block and page caches: goo.gl/mwlzys #magento2 #realmagento

143 days ago
Here it is - the #JohnLewis #Christmas advert for 2016: bit.ly/2eUwfNj

FROM THE BLOG

Mar 10
Charging correct tax on shipping with Magento
By Robert Egginton
If you are reading this then you may be concerned whether you are charging tax correctly for shipping.
Read more

Jan 13
Programatically clearing Magento 2 block and page caches
By Robert Egginton
Magento 2 makes extensive use of both block and full-page caching.
Read more

CONNECT TO US

PARTNERS

Magento sage pay amazon web services Atlassian

© 2017 C3 Media Ltd - All rights reserved.
Registered in England and Wales: Company #6994290 VAT #989029858
0800 1488 234
hello@c3media.co.uk