MANUEL WEISS
COFOUNDER OF CODESHIP

Efficiency in Development Workflows

CODESHIP

About the Author.

**Manuel Weiss is cofounder of Codeship.
He founded the company in 2011 with the vision to help
software development teams across the world become
more efficient.**

Codeship is a hosted Continuous Integration and
Delivery Platform running on Docker that focuses on
speed, security and customizability.

Learn more about Codeship here.

CODESHIP

Table of Contents.

ABOUT THIS EBOOK

Throughout this book you will find blue links from time to time. All those links are clickable and will lead you to pages with further information. For additional content about Development Workflows, Continuous Integration and Continuous Delivery be sure to check out the Codeship Blog at https://blog.codeship.com

Introduction

This ebooks covers all steps the Codeship team takes from deciding on a new feature to shipping a first and following versions. We've been using this workflow, which builds on top of Continuous Deployment, to ship lots of changes per day to our customers for a long time. It has increased the speed with which we can ship new features and drastically lowered the problems we face in production.

This book is especially geared towards development team leads, but definitely has some new valuable information for every developer.

It contains lessons learned through years of iterations on Codeship and thousands of hours spent with other teams and professionals talking about their process and technology. It covers what we have learned about Efficiency in Development Workflows.

Codeship is a hosted Continuous Integration and Continuous Delivery platform. We use it to continuously test and deploy the code we are working on with our team. It is our Testing Server and Deployment platform of choice.

We recommend some tools and services throughout this book. We are in no way affiliated with the companies offering these tools. It's just what worked best for us.

There are plenty of alternatives out there. We encourage you to find out what works best for you.

Developing a new feature

This part focuses on the workflow to implement a new feature. From branching away from master until it is ready for the Pull Request.

Git branching model

At Codeship we follow the GitHub-flow model of development (click to check out Scott Chacon's article), so whenever we start a feature we create a feature or bug branch. Most of our team uses git-extras by visionmedia for this.

```
COMMAND LINE

bug/intercom-cdn
bug/overhead-doc
feature/better-script-deployment-docs
* feature/vagrant-hstore
master
/home/manny/workspace/mothership(feature/vagrant-hstore|√ ) %
```

When the team still was smaller, only one person worked on one branch. If we needed more people to work on a feature we broke it down to the smallest possible chunk.

Take this example: Ben implemented a first basic
version that allowed to invite collaborators from GitHub
to Codeship. He worked on his own feature branch and
had a simple UI that was ready to be shipped. After the
feature passed the Pull Request and Code Review his
changes were shipped.

Teams

⚙ Organization Settings

Managers
managers · 14 Members · 17 Projects

Owners
owners · 3 Members · 17 Projects

Both, Ben and Alex, committed and pushed regularly
while still working on the feature. When a small piece is
done we typically push it to Github to run our complete
test suite.

While Codeship runs all of the integration tests we keep
working on the feature.

This way we very quickly see if our changes broke any
part of the application without running the full test suite
locally. And breaking a feature branch is absolutely ok.

We want our developers to push early and often and let the Testing Server take care of the tests so they do not waste time.

There are numerous advantages in shipping a minimum viable feature first. We keep waiting times between the developers to a minimum while still shipping improvements very quickly. Thereby we remove a lot of unnecessary communication. And we never run into any kind of merge problems when two developers work on the same feature.

REMEMBER

Breaking a feature branch is absolutely ok!

Of course there are challenges with this workflow. Sometimes features are shipped with the expectation that they are improved right afterwards, but something

else needs immediate attention. This way the improvement can take a bit until it is shipped.

Therefore getting that minimum viable feature right is very important. Big enough to be valuable, but small enough to be shipped fast.

Pull Requests and Code Review

In the previous part we explained how we start with a new feature and the workflow to implement it. This part focuses on the steps necessary to get your feature ready to be shipped.

Using Pull Requests

As mentioned we use GitHub Flow on GitHub. But the whole workflow we talk about in this ebook is also possible when working with Bitbucket or other Source Code Management services.

We do not have a policy when a Pull Request should be opened. Some of our developers open them when they start a feature, some wait until the feature is implemented. We then push regularly to that branch as explained a few pages earlier.

Open Pull Requests are helpful as everyone can see what the other members of the team are working on. One really important part of feature branches and Pull Requests are proper commit messages.
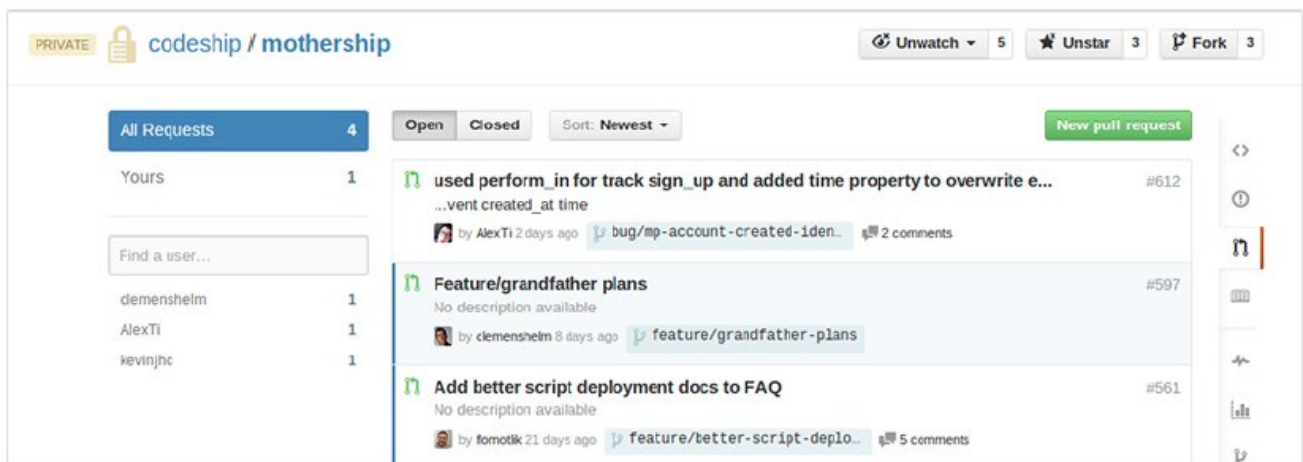
Writing good commit messages

As a distributed version control system, git makes it easy to take large changes and split them into small commits. These commits have a descriptive message with a short explanation in the first line and, if necessary, a longer description after a separation line. The git documentation project has some nice guidelines on it. Giving every commit a proper message and splitting large changes into a number of small commits makes your code history easier to read. It will be easier for your teammates to understand the history of your code.

We prefer git-gui for commiting and splitting up

commits. It does only this job, but it does it very well. Of course there are some other git tools. Find out which you like best.

Using GitHub issues and labels

Once the feature is ready to be reviewed we assign its Pull Request to somebody else on the team. You have to read through the changes on GitHub at least once before handing the Pull Request. Making sure no obvious errors are in the code reduces the time and cycles necessary for a good code review.

It is very helpful to inspect the changes outside of your own editor and with the excellent GitHub comparison view.

```
8    8        include Codeship::Helper
     9  +      include Codeship::Retryable
9    10
10   11       def perform
11   12         logger.info "Checking to restore cache for project #{config.project_id}"
12   13         cache_store = CacheStore.new(config.project_id)
13      -        cache_store.download if cache_store.remote_exists? && (!cache_store.local_exists? || cache_sto
     14 +
     15 +        retryable(tries: 2) do |retry_count|
     16 +          logger.info "Downloading Cache File (project: #{config.project_id} attempt: #{retry_count})"
     17 +          cache_store.download if cache_store.remote_exists? && (!cache_store.local_exists? || cache_s
     18 +        end
```

Now that the Pull Request is assigned, somebody on our team has a GitHub issue with his or her name attached. This way anybody can easily filter out the Pull Requests that need to be looked at.

The last step is labeling the Pull Request with needs review. This way the reviewer knows this Pull Request is ready to be reviewed.
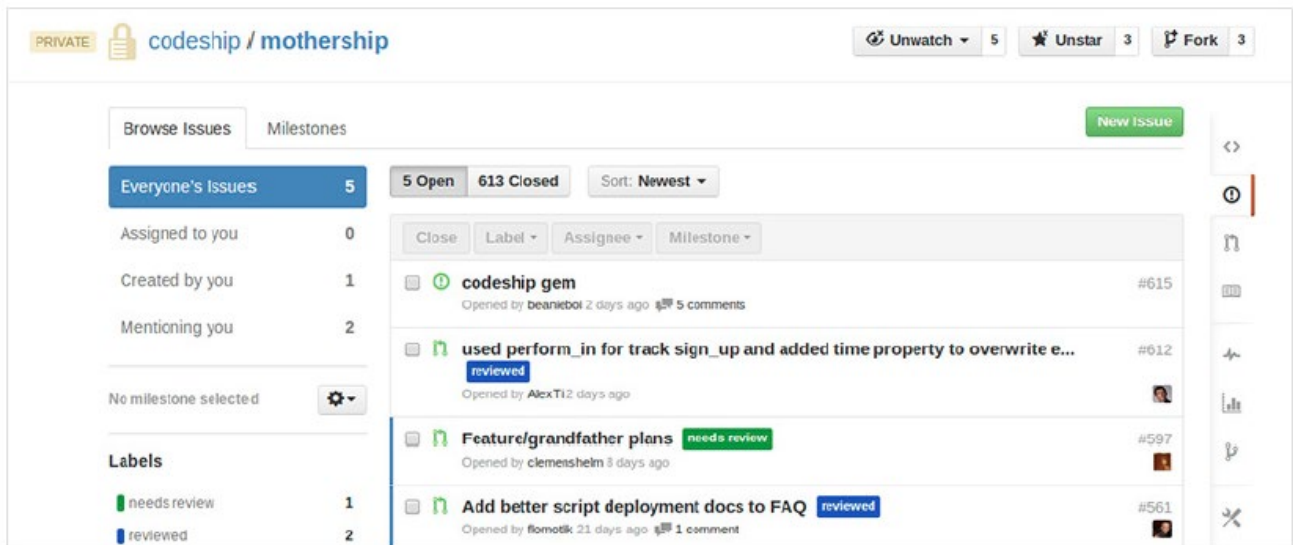
When starting a Pull Request, the developer and reviewer can decide on looking through the Pull Request together or not. Especially for large changes reviewing the code as a pair is great.

We review the structure and syntax of the code, but we currently do not review the functionality itself. Before implementing a feature we discuss it in detail, so there is a common understanding. Once the feature is implemented, we trust our engineers that it does what it should. We usually don't do feature reviews on our staging system.
Of course you can do functional reviews for large and public facing features. Just so a second pair of eyes looks at changes that possibly impact a lot of users.

When the review is done it either gets merged or, if the reviewer leaves comments, the GitHub issue is labeled as reviewed. This way developers can take a look at their open issues and see which have been reviewed and which need more attention.

The feature is ready to be merged as soon as the developer either updates the Pull Request or explains why she did it differently.

There's one challenge with every release workflow that involves code reviews: Getting the team to review as fast as possible without interrupting their work. We try to do it asynchronously for smaller features and sitting down for shared code review on larger features. This is no silver bullet, but in our experience it works well.

REMEMBER

It is very helpful to inspect changes outside of your own editor.

GitHub Merge Button

To make sure nobody introduces changes into our production system without review, we have a strict rule that you can never merge your own code. This rule is not enforced through technology, but a team policy.

The only exceptions are scheduled releases, for example a maintenance deployment. If this happens we let somebody review the changes and leave a ready to be merged comment. The build can then be merged whenever the developer wants to.

By releasing through the GitHub merge button everyone on the team can push to production. Making this possible for every developer or designer in your team is very valuable. It allows everyone to take part in the process and it speeds up the daily work.

You can add more commits to this pull request by pushing to the **bug/mp-account-created-identify** branch on **codeship/mothership**

Good to merge — Build succeeded (Details)

This pull request can be automatically merged.
You can also merge branches on the command line.
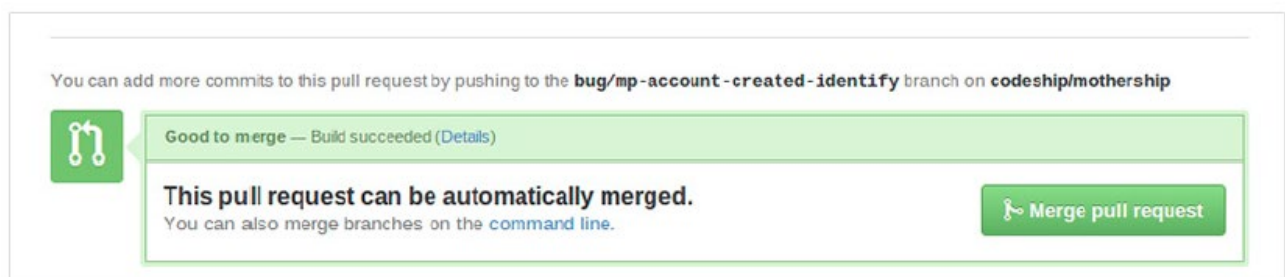
⌥ Merge pull request

As soon as we hit the merge button our tests are kicked off on our Testing Server and the deployment starts.

Deployment Pipelines and Zero Downtime Deployment

In the previous part we talked about how we review code, open Pull Requests and use Github issues to manage our Development Workflow.

This part will show every step that happens after a Pull Request is merged into our master branch. We use an automated deployment pipeline for releasing our code into production.

Deployment Pipelines

A Deployment Pipeline lays out the whole process that your code needs to go through from your repository to production. It breaks the build into several parts (e.g. build, test and deploy) and all the associated steps that need to be taken. By defining a pipeline it is always clear which step needs to happen next. Martin Fowler describes it really well in his blog post.

If you want to dig deeper into Deployment Pipelines we recommend Jez Humble's and David Farley's book: Continuous Delivery.

Configure deployments per branch

To automate deployment to different environments we have found that it works best to define actions for different git branches. If you always push the latest commit of your production branch to your production environment, it is very easy to determine what is currently deployed by just looking at the git branch. Git and other source code management systems only permit one commit at the top of a branch, so there can be no confusion.

At Codeship we deploy our master branch automatically to production. Many of our customers deploy the master branch to a staging environment and a production branch to their production environment. A simple git merge and git push, or a Github Pull Request, is their way of releasing their changes.

One problem with this approach is that branch names have to be meaningful. Having a development branch which is deployed to staging and a master branch that gets deployed to production can confuse new team members. Naming branches that get deployed "production" or "staging" is more intuitive.

"Master" is a convention in git and should be kept, but dedicated branch names are easier to understand in a Deployment Pipeline.

Deployment Strategy

As soon as the feature branch is merged into our master, a new build is started on our Testing Server. We run the same test commands as we did on the feature branch to make sure there are no problems in the merged version.

When all tests pass for the master branch the deployment starts. When the team was still smaller we also checked if all database migrations were working.

First we deployed to staging. Then we ran our set of migrations. We copied our production database to staging once a day (click to read a blog post about how we did this). Therefore, when we ran migrations on staging, the database was very close to our production database. This allowed us to make sure our migrations worked, before deploying to production. Today, our production database is just way too big and it would take too long to copy it every day, so we are skipping this part nowadays.

The last step in our staging deployment is calling the URL of our staging site to make sure it started successfully.

Wget and its retry capabilities make sure the website is up and running. Afterwards we run our migrations in a separate step after the deployment and clear our cache.

We then repeat the whole process for our production system. We push to the Heroku production repository and check that the site still works.

An enhancement would be to have tests that run against the deployed version, but so far we haven't had any problems without these tests. Our extensive test suite has caught all problems so far.

Zero Downtime Deployment

As we want to deploy multiple times a day without any downtime we use Heroku's preboot feature. Whenever we push a new release, it starts this release on a second server and switches the routing to it after about 3 minutes.

The downside is that Zero Downtime Deployments require more care with database changes. As two versions of your codebase need to be able to work at the same time you can't just remove or rename fields.

Renaming or deleting a column or table needs to be spread out over several deployments. This way we make sure that the application still works with every incremental change.

It is important to automate every step of the deployment.
No matter if you want to deploy your code on every
merge to master or trigger it manually by merging the
master into another branch.

Conclusion

Now that you've seen our workflow in detail, the main
take away we want you to leave with is that process and
thinking about process is the lifeblood of every team.

Technology comes and goes, but through a great
process you can build something amazing. Taking these
steps to eventually come to Continuous Deployment or
Continuous Delivery will make you and your team more
productive, successful and in the end also happy.

If you want to learn more about Continuous Delivery you
should check out my free 5-Day Continuous Delivery
Email Crash Course.

If you have any additional questions feel free to contact
us and consider following us on twitter @codeship

Further Reading and Resources

▶ Scott Chacon's article on GitHub Flow

▶ git-extras by visionmedia

▶ Github App for Mac

▶ Sourcetree – free Git & Mercurial client for Windows or Mac

▶ tig – Text-mode interface for git

▶ Gitk – The git repository browser

▶ How to set up BitBucket on the Codeship

▶ Continuous Delivery: Book by Jez Humble and David Farley

▶ The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win

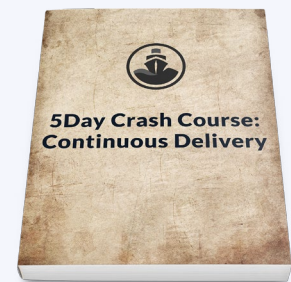▶ Visible Ops Security: Achieving Common Security and IT Operations Objectives in 4 Practical Steps

More Codeship Resources.

How Continuous Delivery makes you more productive.

Continuous Delivery, and the workflow that goes along with it, helps create a better product faster.

Learn with this email course

Continuous Integration Essentials Guide.

How CI ties in with Continuous Deployment and Continuous Delivery and how to get started with it.

Download this eBook

Why Containers and Docker are the Future.

Learn about the rise of the Container Stack and why Docker and its ecosystem play such a big part in it.

Download this eBook

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.

🚀	Works out of the box
⚙️	Preinstalled CI dependencies
🖥️	Optimized hosted infrastructure
🚀	Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.

⚙️	Customizability & Full Autonomy
🖥️	Local CLI tool
🖥️	Dedicated single-tenant instances
🚀	Deploy anywhere

LEARN MORE