Praekelt Ways of Working Documentation

Release 1

Praekelt Foundation devs and individual contributors

Sep 07, 2017

Contents

Hey, hopefully you're here because you're about to embark on a project with us.

Here are some bits & pieces of documentation to help you get up to speed with our ways of working, the tools we use, processes we follow and the things we expect from you when doing projects.

If anything is unclear or plain wrong please do notify us, or better yet submit a pull request to the repository!

Enjoy!

CHAPTER 1

Engineering teams

Engineering is split into 3 teams.

Health

The health team works on:

- MomConnect South Africa
- HelloMama Nigeria
- FamilyConnect Uganda
- NurseConnect South Africa

Youth

The youth team works on:

- Springster
- Tune Me
- The Internet of Good Things
- BabyCenter

Site reliability engineering (SRE)

The SRE team is responsible for hosting and infrastructure.

Because Praekelt deals with health and medical information, we normally use infrastructure hosted inside the countries where we operate:

- South Africa
- Nigeria
- Uganda
- Amazon Web Services eu-west-1

CHAPTER 2

Projects

HelloMama

HelloMama is a project that sends messages to pregnant and new mothers in Nigeria.

It's hosted in Nigeria.

It runs the Seed stack.

The hub app for HelloMama is hellomama-registration.

MomConnect

MomConnect is a project that sends messages to pregnant and new mothers in South Africa. It's a programme by the South African National Department of Health (NDoH).

It's hosted in South Africa.

It runs the Seed stack.

The hub app for MomConnect is ndoh-hub.

The JavaScript sandbox apps for HelloMama are in ndoh-jsbox.

Springster

Springster is a Girl Effect project publishing information for adolescent girls. It's available in 65 low and middle income countries through Facebook's Free Basics platform.

It's online at http://za.heyspringster.com/ and is hosted in South Africa

Tune Me

Tune Me is a mobile site for providing health information to people in Botswana, Malawi, Namibia, Swaziland, Zambia and Zimbabwe.

It's built using Molo and is hosted in South Africa.

CHAPTER 3

Infrastructure

Nigeria

In Nigeria we use [hosting provider A] running [container scheduler B].

South Africa

In South Africa we use [hosting provider A] running [container scheduler B].

Build

Build containers using Travis.

We used to use Sideloader but that's deprecated.

Deployment

Deploy containers using Mission Control.

Monitoring

CHAPTER 4

Tech

This section documents the technology we use for the majority of our engineering work.

Using anything not documented here may have implications for hosting and infrastructure and needs to be approved by engineering management before work starts.

Django

We use the Django Python framework heavily.

You should use the latest stable release of Django *as of the start of the project* unless otherwise specified. Pin that version in your requirements so that the project won't break by accidentally being deployed on a newer version without testing.

You should build Docker containers for your Django app using django-bootstrap as a base:

```
FROM praekeltfoundation/django-bootstrap:py3
```

HTTPS

HTTPS secures information when it's being transmitted over the internet. Everybody should use HTTPS, but it is especially important that we use HTTPS because of the sensitive data we transmit.

Compatibility

Some of this will be trial and error. Some settings can be tweaked on our side but some particularly old devices/software will just never work with our system.

It's difficult to anticipate what devices are out there being used in large numbers, and what browsers are being used on those devices. In a number of cases devices with old operating systems (like Windows XP, old Android and old BlackBerry OS) will not be compatible with our HTTPS setup when using the default/built-in browser. If users have

a recent version of a commonly used browser like Google Chrome, Mozilla Firefox, or Opera Browser installed then that should be compatible.

Users accessing an HTTPS site via a recent version of Opera Mini should not have any compatibility issues on any platform but the effectiveness of the encrypted connection is reduced as the connection is decrypted at Opera's servers.

You can use the SSL Server Test from SSL Labs with one of our domains to see the most up-to-date configuration that we support.

Server Name Indication (SNI) compatibility

The first and most important feature that any device connecting to an HTTPS site on our cluster will need is Server Name Indication (SNI) support. This, unfortunately, cannot be worked around and is a strict requirement born out of the fundamental design of the cluster. Please see the Wikipedia article for full details.

The native/built-in browsers on the following mobile devices do not support SNI (although it may be possible for users to install a browser that does support SNI):

- Android 2.3 "Gingerbread" and earlier
- Nokia Symbian OS
- BlackBerry OS 7.1 and earlier

The following browsers on desktop devices do not support SNI:

- Internet Explorer on Windows XP

It's also important to note that the following Python software we use does not support SNI:

- The `ssl` standard library module in Python < 2.7.9
- Vumi < 0.6.15
- Twisted < 14.0.0

Transport Layer Security (TLS) compatibility

The load balancer is currently only configured to support some versions of Transport Layer Security (TLS), namely TLS 1.0, 1.1 and 1.2.

Some older browsers (Internet Explorer 6 or below) do not support our configuration. Once again, Wikipedia has a nice table detailing compatibility.

It's generally a good idea from a security standpoint to only support the latest TLS versions. We will never support technologies older than TLS 1.0 such as SSL 3.0. We may have to disable the older versions of TLS that we support as new vulnerabilities are found.

Enabling HTTPS on Mesos clusters

The SRE team have made it really easy to set up HTTPS with free, automatically renewing certificates from Let's Encrypt.

Rate limits

Let's Encrypt is a free service but ultimately the service has to have some limits. Please read the Let's Encrypt documentation for full details. The only limit that will be relevant to most people is the "Certificates per Registered Domain" limit. This limit is set at **20 certificates per week**.

The way this limit works is important to understand: it is applied per registered domain. This means that it is easy for us to hit the rate limit if we issue certificates for many sites using a wildcard domain under one of Praekelt's commonly used domain names. For example, a domain like `seed-message-sender.seed.ng.p16n.org` is actually under the registered domain `p16n.org` - which means it shares its rate limit with many other Praekelt sites.

tl;dr: If you have a domain that has been bought specifically for your project then you probably don't need to worry about rate limits. If you are using a domain that ends in `p16n.org` or `unicore.io` then please speak to SRE before trying to issue a certificate. Certificates are free but not unlimited.

Setup

Add an application label to the app you want to set up using Mission Control. Right now only one domain per app is supported.

The label is called `MARATHON_ACME_0_DOMAIN`. Set it to the domain name you want a certificate for.

A few moments after adding the label the application should be available over HTTPS.

Redirecting HTTP to HTTPS

Once you've verified that the application works correctly over HTTPS, set the label `HAPROXY_0_REDIRECT_TO_HTTPS` to `true` to redirect all HTTP traffic to HTTPS.

Using HTTP Strict Transport Security

Another option that you can add for enhanced security is to enable HTTP Strict Transport Security (HSTS). This option improves security when your site is accessed via a modern web browser. It won't have any effect if your app is an API rather than a website.

You **must** verify that the certificate and HTTP to HTTPS redirect is working correctly before enabling HSTS. Once a user's browser "sees" the HSTS header for the first time **it will only connect to the site via HTTPS for the next 6 months**. This is a thing that is easy to mess up but if your site has been running with HTTPS and redirects enabled for a while without issues then enabling HSTS is a good idea.

To enable HSTS, set the application label `HAPROXY_0_USE_HSTS` to a value of `true`.

The HSTS header that is sent does not include the `includeSubDomains` or `preload` directives.

Junebug

Junebug is an HTTP API for sending messages. It provides an API for configuring vumi transports.

Junebug follows our standard approach for publishing Python packages.

Molo

Molo is a content management system built on top of Wagtail.

The source is on GitHub.

There are plugins to provide additional functionality when needed:

- https://github.com/praekelt/molo.profiles

- https://github.com/praekelt/molo.yourwords
- https://github.com/praekelt/molo.polls

Python packages

A lot of our code is published as Python packages on PyPI.

Development

Make changes to the repository that contains the Python package using the normal GitHub flow.

Bump the version in the repository. You might have to look in:

- `setup.py`
- `package/__init__.py`
- `docs/conf.py`

Release

Use `git flow release` to publish a new release to GitHub. This will create a new tag which will cause Travis to build.

The `.travis.yml` file will deploy the package to PyPI as the `praekelt.org` user.

For many package repositories, there's another repo prefixed with `docker-`. For example, `junebug` and `docker-junebug`.

pyup.io will make an automatic pull request to the `docker-` repo to increment the `requirements.txt` file.

When that pull request is merged, Travis will build a container image and deploy it to Docker Hub as the `praekeltorgdeploy` user to the `praekeltfoundation` namespace.

Deployment

Use Mission Control to deploy the Docker Hub image to QA and production.

Seed stack

The Seed stack is a set of microservices:

- `seed-control-interface-service`
- `seed-message-sender`
- `seed-scheduler`
- `seed-stage-based-messaging`
- `seed-auth-api`
- `seed-control-interface`
- `seed-identity-store`

- `seed-service-rating`

We build containers for the Seed stack.

Each deployment of the seed stack has a "hub" or "registration" app which contains logic specific to that country or environment.

Vumi

We build on top of our Vumi platform for SMS, USSD and other messaging protocols.

Vumi is a scalable messaging engine which we use for SMS, USSD and other messaging protocols.

We build Vumi in a container and push it to Docker Hub.

Nowadays for the most part Vumi is only used via Junebug.

JavaScript sandbox

Apps can be written for Vumi to power messaging campaigns or information systems. These apps can be written in JavaScript to run in a sandboxed environment (which is our preferred option) or in Python.

Vumi Go

Vumi Go is a hosted version of Vumi. Where Vumi gives you the tools to build large scale messaging applications, Vumi Go provides you with a working environment that is already integrated into numerous countries.

Vumi Go is deprecated and will not be available after the end of 2017.

CHAPTER 5

Tools we use

The following are tools we use on a regular basis and which we expect you to either use or at least be very familiar with.

IRC

IRC is our team's communication tool of choice. Join us in `#prk-dev` for general developer support, or `#vumi` or `#jmbo` for development of those platforms, on irc://irc.freenode.net/.

Various tools report into these channels and provide insight into what is going on.

Git

We use Git. If you work with us, you will use Git for revision control, and GitHub. There is no exception.

Provide us with your GitHub username and we will provide you with a repository to work on. All repositories are to be hosted under the Praekelt Organization on GitHub.

Please read What's in a Good Commit? for a good introduction to effective use of version control commits.

Avoid these:

- Don't commit merge conflicts. See the Pro Git book on merge conflicts
- Don't commit snapshots. Only make one change per commit. **Read What's in a Good Commit above.**
- Don't commit large content files. Manage them in the CMS.

git-flow

We use the git-flow branching model as part of our development. It's a convenient way to manage your branches. You are not required to use Git Flow but you are required to follow naming conventions it sets with regard to branch names

and prefixes.

Have a read through the blog post describing the general idea and follow the installation instructions in the repository to install it for your development platform of choice.

Unless you've explicitly been told otherwise, we require our team to review your code before landing it in the develop branch. Please provide pull requests for our review, the command line tool Hub (see below) is a convenient way of turning GitHub issues into pull-requests.

The pull-request requirement still remains when using Jira. You can still use Hub - however your Jira ticket's status will not automatically change when the feature branch lands, so you will need to update this yourself.

Please read Useful Github Patterns to see ways of working with branches and pull requests that we like.

HubFlow

Hubflow is an adapted version of Gitflow, specifically tailored for use with Github.

It provides the usefulness of git-flow with Github goodness embedded.

For more information on that, see this link: Hubflow

Hub

For projects with issues tracked in Github issues, We use Hub to interface with GitHub's API. It allows one to turn issues on GitHub into pull-requests. If that is done then once the pull-request is merged into the main branch the issue is automatically closed.

Issues & Tickets

For project work we use Jira. Only our core open-source platforms maintain their issues in the GitHub repository.

You will be given an account to use which will have access to the relevant projects.

For development, if there is no ticket it does not exist. Make sure the work you are doing has a ticket and is being tracked. Stop working and protest immediately if people are treating your mailbox as their ticketing system. We've tried that, it does not work.

If a Jira project has a workflow, you need to update your tickets appropriately: New -> Open -> Fixed in dev (when pushed to github) -> Deployed to QA

Our QA team will move the ticket to QA Passed, and our DevOps team will be responsible for the production deployment before the ticket is resolved.

If a ticket is QA Failed then it's back into your section of the workflow.

A ticket should represent a solid piece of work you intend to do. Make an effort to keep the work you are trying to do in one ticket to no more than 16 hours.

Any estimate you make for actual work done beyond 16 hours is assumed to be

1. largely thumb-suck.
2. going to be very hard to review.

Make an effort to keep it to 16 hours or break it up unto multiple tickets each representing 16 hours of work.

Sentry

We have a dedicated Sentry instance for our projects. You are expected to configure your application to make use of this for error reporting.

You will be given access to your Sentry project and access tokens to will be made available for you to configure your application's client with.

Puppet

We try and automate as much as possible, this includes our hosting environment. You will need to give us your SSH key so we can provision a machine for your project. Generally you will be given access to a machine that is to be used for QA. Since our DevOps team do the production deployments, and you will get access to production error reports via Sentry, you won't get access to production without a valid need for troubleshooting, and then it will be without sudo access.

These machines are provisioned using *Puppet*. You will not get access to our puppet repository. If you need specific software installed on your machine that it was not provisioned with then please ask for it to be added. Do not install it yourself without notifying us. This would break our assumption that every machine can be provisioned from scratch with puppet.

If the machine you've been working on needs to be rebuilt and you've made changes that are not in puppet then it'll be provisioned without those changes.

Sideloader

Our DevOps team automate deploys using Sideloader, our tool that creates deb packages from github repos. To enable a repo for this deploy automation, create a .deploy.yaml file in your repository, listing dependencies and scripts.

We then use puppet to install the debs whenever a new one is published. Ask our DevOps team for help with Sideloader, and to set up the puppet automation to install the debs.

We can optionally set up a post commit hook to deploy any changes that are pushed to the develop branch, to QA - if you're feeling lucky...

See Sideloader help for more info (requires login via github).

Databases / data stores

We use the following services to store our data. Not all projects will use all of them but generally a number of these will be involved.

1. PostgreSQL
2. Riak
3. Memcached
4. Redis
5. Neo4J

These will be made available to you on a per project basis. Puppet ensures that each of these are backed up.

Django Applications

For Django applications, some applications are mandatory:

1. Sentry for application reporting.
2. Django Migrations for handling database schema changes
3. Nose for running tests.
4. Haystack for search.
5. Memcached for caching.

We strongly recommend you use our sample django-skeleton as a starting point for Django projects, as it has some of these already included.

Translations

We use Gettext or translations in shell scripts, applications and web pages. Read more about Gettext along with some examples on Wikipedia: http://en.wikipedia.org/wiki/Gettext

In Django, Gettext is used by default for translations, utilizing ugettext_lazy for models.py and ugettext in other places. We like {% trans %} and {% blocktrans %} tags and enforce these for our open source products.

Graphite

We use Graphite for the majority of our metric publishing for dashboards. If appropriate, you will be given details for the Graphite server and how metrics are to be published to it.

Front-end

Sass CSS pre-processor so that we can take advantage of things that CSS doesn't have yet, or doesn't do properly: variables; nesting (used sparingly); CSS partials / includes; media queries used more like element queries; mixins.

JavaScript task runners like Grunt and Gulp, with lots of plugins. These handle code linting, image minification, processing Sass into CSS, concatenation and minification of CSS and JS, and running tests.

CHAPTER 6

Tutorials

This section contains a deeper look at some of the technologies that we use and how we use them.

Creating Tagged Releases

This tutorial covers creating Tagged Releases with Github, Travis and Mission Control.

The ability to create tagged releases in our molo sites, allows us to test a QA version of our code, without merging our changes into the develop branch. Note that this currently only works for sites like TuneMe and Springster, not molo.core or a particular package.

This tutorial will explain how to create a tagged release for a branch on TuneMe, as an example.

Creating your first tagged release

Ensure everything is configured

Make sure that Travis is set up for to create tagged releases. See this commit as an example of how to do this. Check that the account is set up to handle tagged releases on Docker Hub. At the time of writing this, both Springster and TuneMe are set up to handle tagged releases.

Create the tag

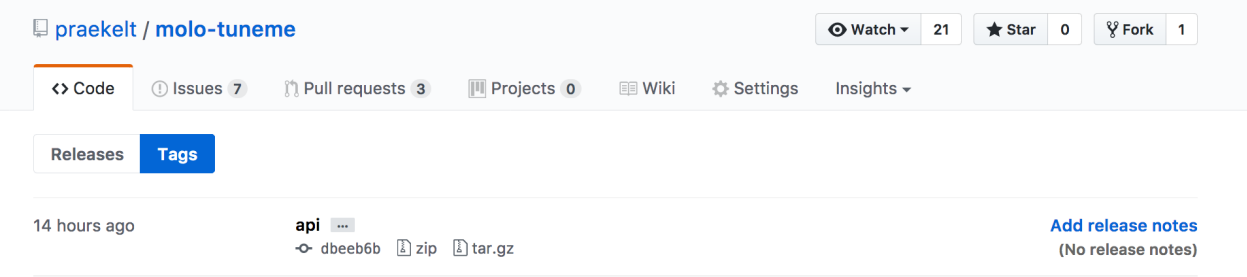
Create the tag locally, on the branch that you want to test in a QA environment. The tag name should be something short and descriptive. For example `api`.

```
git tag <tag name>
```

Tell github about the tag by pushing your tag

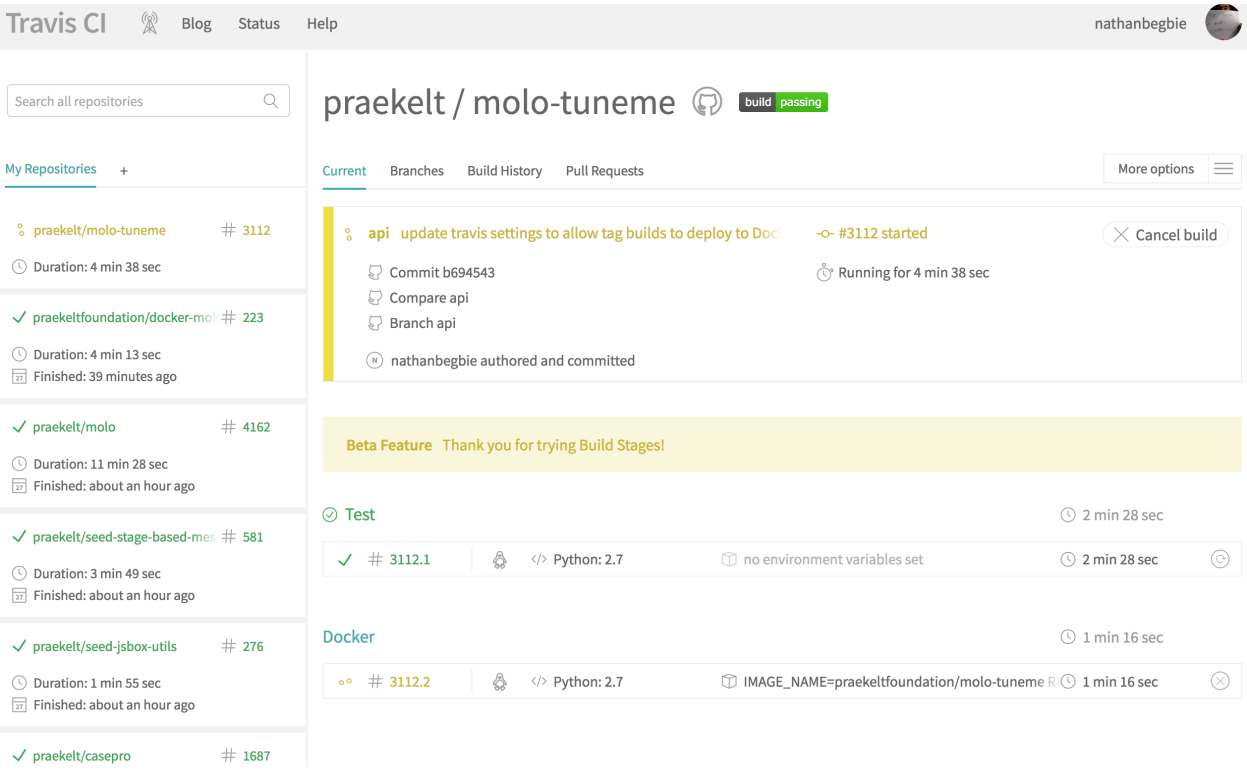
```
git push origin <tag name>
```

You can check that that tag has been created by going to the `/releases` section on Github. e.g. `https://github.com/praekelt/molo-tuneme/releases` or more specifically for the tags, `https://github.com/praekelt/molo-tuneme/tags`.

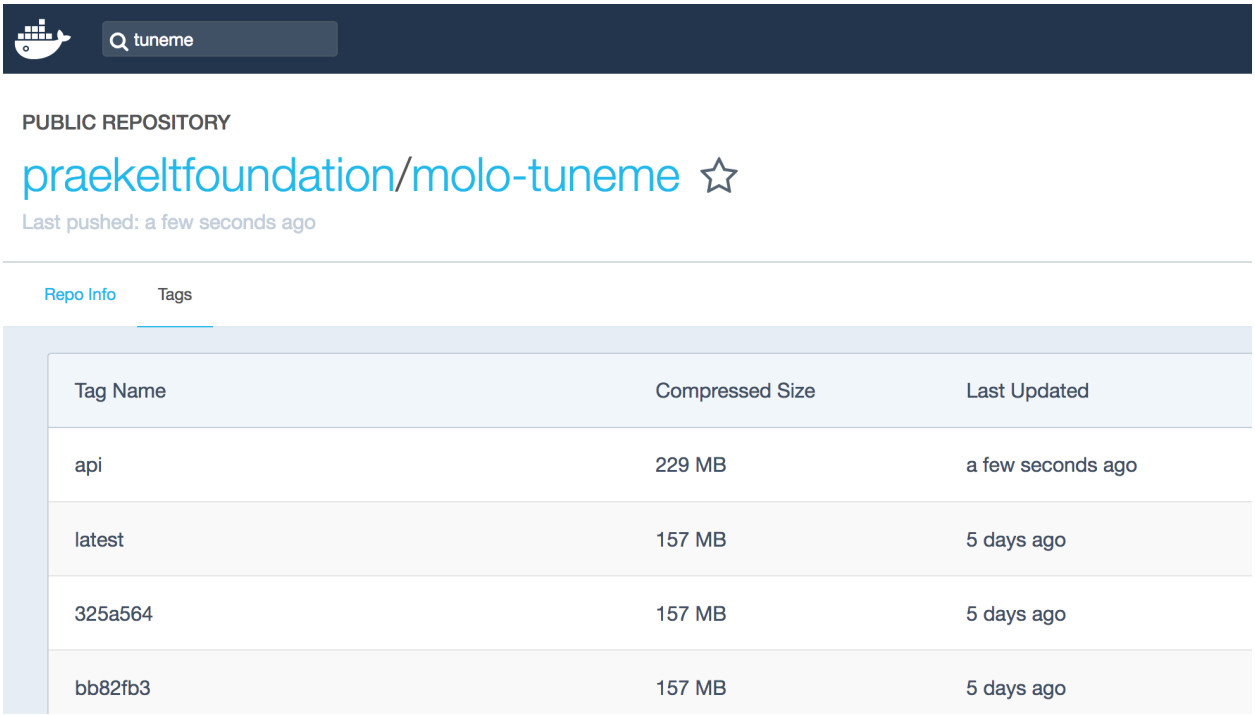


Check on the Docker Image creation process

If you check Travis, it should automatically start a new build, named after your tagged release.

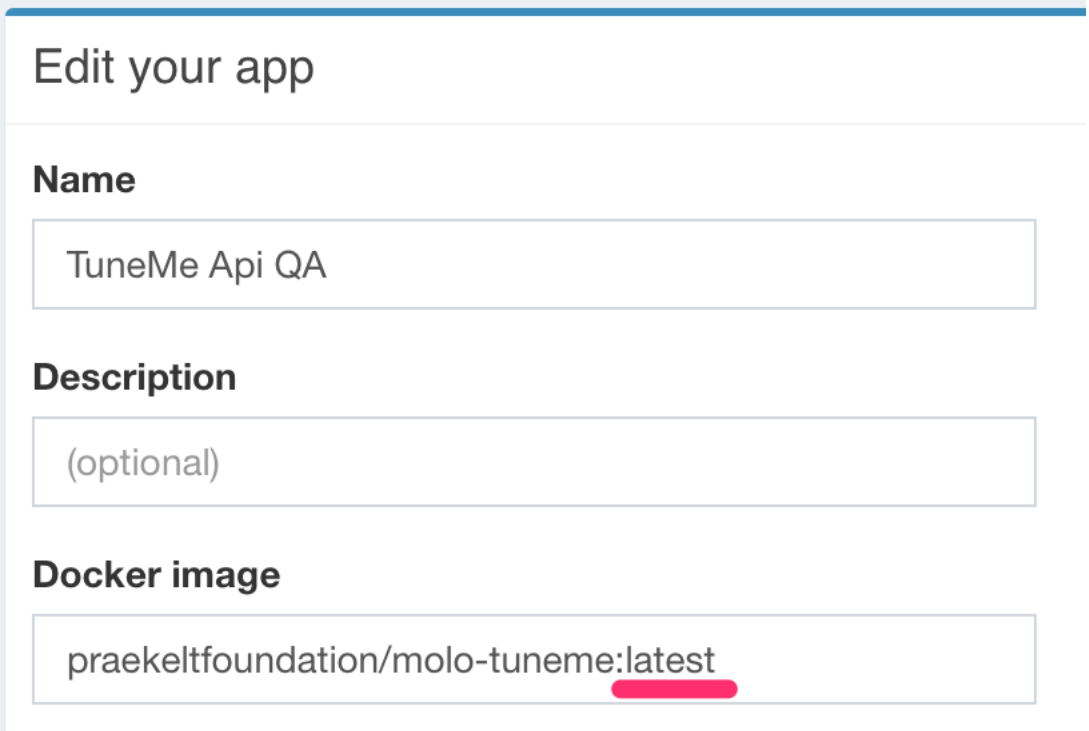


Once all of the tests have passed on Travis, it will bundle everything up and create a Docker Image, containing everything needed to run an instance of your code in QA. That image will be tagged with the same tag you used earlier. If everything passes, the Image will then be pushed to the associated account on Docker Hub. In this example, it will be `https://hub.docker.com/r/praekeltfoundation/molo-tuneme/tags/`. You should see the tag you created, listed at the top.

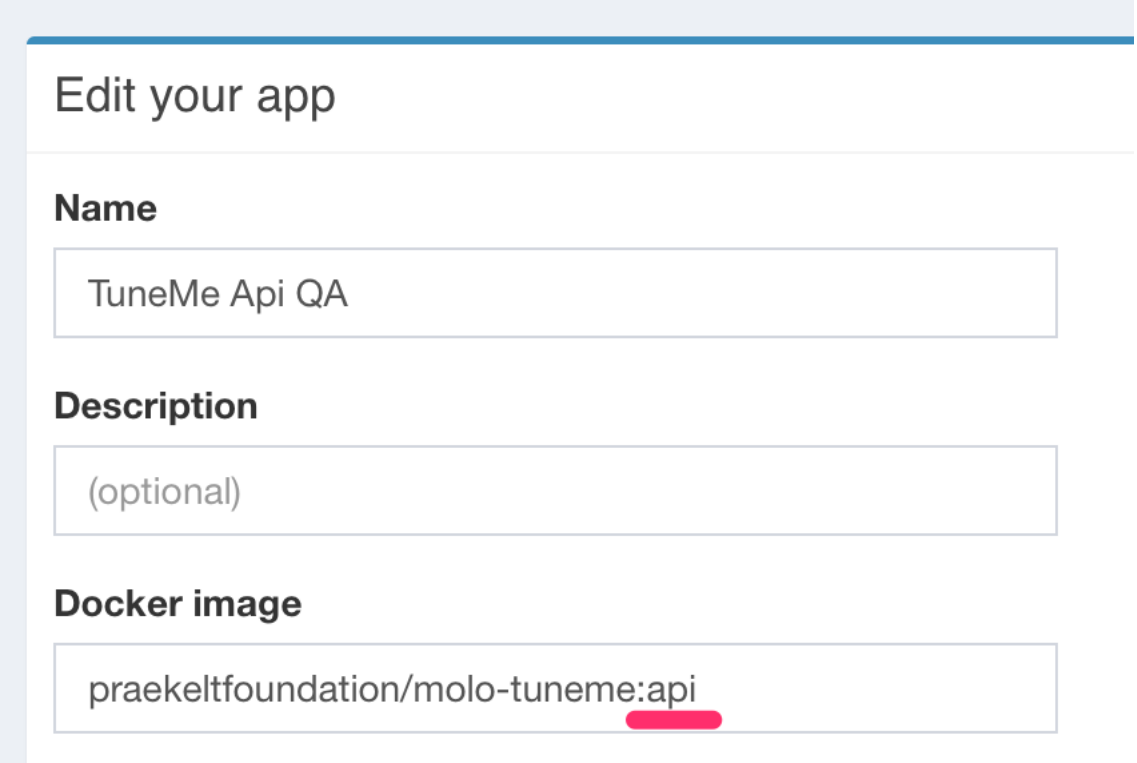


Update QA Mission Control

Once you see the tag, Go to QA Mission Control, find the QA instance that you want to test with. Click on the 'edit' link. Check the field called `Docker Image`



Change the listed docker image being used to your tag. In this case it will be `api`.



Finally, hit 'Submit' at the bottom of the page to ensure the changes are made. The site will restart and you should see your changes when it gets back up and running.

Warnings

One of the possible gotchas with QA instances is the possiblility of corrupting the DB with migrations. Speak to someone if you're not sure what affects your changes could have.

Recreating a tagged release

Okay, so you've made some changes and you want to see those new changes in QA.

Delete the tags

In order to create the new tagged release, you need to delete the tag where it is being used. To do this, first delete the tag on Gihub.

You can do this via the command line using:

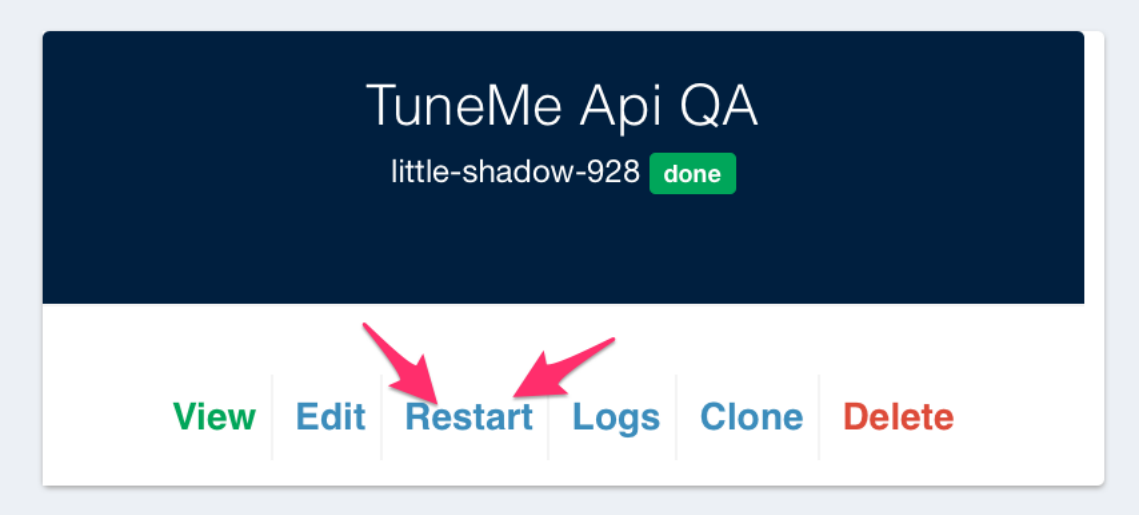
```
git push --delete origin <tag name>
```

Then delete the tag locally, using the following:

```
git tag --delete <tag name>
```

Follow the same process as above

You then need to recreate the tags, push them to Github, wait for tests to pass and Docker Hub to update. You don't need to update the Docker Image on Mission Control. Instead, just hit the restart button for the QA instance on MC.



Conclusion

Congrats! We can now test our code in a QA environment before commiting our code to the develop branch. Note that if your work is being done on a package, it needs to have a release created before you can test on a QA site.

CHAPTER 7

Intellectual Property

All code produced in exchange for remuneration by a Praekelt company must have copyright assigned to "Praekelt Foundation" or "Praekelt Consulting" as appropriate.

All code published into open Github repositories must be licenced under the BSD 3-Clause Licence. Any third party open source modules used for the project must be under an explicit, compatible licence, and if belonging to you as a development partner, must exist prior to the start of the project to remain your IP.

Code snippets used from other sources (i.e. not complete files, or single files lifted from other projects) must be attributed in the header of the file in a comment, including URL of the source, and author.

CHAPTER 8

Our project process

The lifecycle of our projects is typically as follows:

1. We produce a Scope of Work for a project, which might not have all the technical details, but should be comprehensive enough to list all the features so that you can quote on the project's development. Wireframes may also be provided as part of the scope for your CE (Cost Estimate).
2. We work on a fixed cost basis for a fixed scope. If the scope changes, we ask you for a costing for the delta or new work.
3. The authorisation to proceed with work consists of a Purchase Order, without which you cannot invoice us - so never start work without the PO.
4. Development commences - see below. **If you don't have a github repo by this point, please bug us until we provide it - please do not use your own repo.**
5. We provide you with one QA server, with the same OS setup that we'll use in production - for all the projects you do for us, unless a project has special needs which justify its own QA server. **Please bug us for a QA URL for this project to be pointed to your QA server.** It must be on our domain for client UAT. Please note that QA may need sample or real data to be populated. Often, the QA data gets migrated to the production site when finally deploying that, so please ensure that dummy data can be cleaned up, and use **CMS credentials** on QA that are *suitable for production*.
6. You are responsible for deploying your code to this QA server, so that you can support the fixing of bugs found during our QA testing. You should **always** deploy to QA from the github repo, to avoid any side effects of uncommitted code.
7. We'll deploy to production so that we can support it - see below.

CHAPTER 9

Our development process

The process involved in how we work is fairly straight forward and we expect you to follow this as well.

1. We use Git Flow's convention with regard to branch names.
2. All work requires a ticket with a unique number or name.
3. Work happens in a feature branch. Feature branches names are composed of the ticket / issue number along with a one-line description of the issue.
4. Write tests for the new features you are developing.
5. Make one change per commit, and follow What's in a Good Commit?
6. Put the ticket number in the commit message. For github issues in particular, see Closing issues via commit messages
7. Your schema changes are expected to be handled by a schema migration script.
8. When work in a feature branch is ready for review then we create a pull-request.
9. All collaborators on the GitHub repository are notified of the pull-request and will start the process of reviewing the changes.
10. Any issues, concerns or changes raised or recommended are expected to be attended to. Once done please notify the reviewers of the changes and ask for the changes to be re-reviewed.
11. Once all the changes are approved and one or more of the collaborators has left a *:+1:* in the pull-request's comments it can be merged into the main branch and is ready for a deploy.

For your code to be ready for review we have the following expectations:

1. It is to be pep8 compliant and pyflakes is not raising any issues.
2. It is to have tests. Unless otherwise agreed, 90% test coverage is required. See Coverage
3. The tests have to pass.
4. There are no commented lines of code.
5. There is adequate amount of documentation.

Example flow

```
$ virtualenv ve
$ source ve/bin/activate
(ve)$ git flow feature start issue-1-update-documentation
(ve)$ git flow feature publish issue-1-update-documentation
..// hack hack hack // ..
(ve)$ nosetests
.............
----------------------------------------------------------------------
Ran 13 tests in 0.194s
OK
(ve)$ git push
(ve)$ hub pull-request -b develop -i 1
https://github.com/praekelt/some-repository/pulls/1
..// review, update, re-eview, update, re-review ... +1 // ..
(ve)$ git flow feature finish issue-1-update-documentation
..// changes merged to develop by git flow // ..
(ve)$ git push
```

CHAPTER 10

Our front-end development process

We build web sites so that people can access them quickly and easily, regardless of the device they're using, the type of connection they are on, or any disabilities they have. That means we build things with **Progressive Enhancement**.

A basic, functional, experience is delivered to everyone; JavaScript is not required for any key functionality. We then do feature tests in JavaScript to load in additional CSS and JS enhancements that we've determined the browser can handle. Different browsers will be served different experience; they will be consistent and may be quite similar, but will not be identical.

We also care about front-end performance and accessibility, so we run regular perfomance and accessibility audits.

CSS

We keep all our CSS in a few, minified, external CSS files. We don't use inline style blocks or write inline CSS.

We write our CSS like SMACSS. CSS is organised into: Base; Layout; lots of Modules; States. We keep nesting shallow, and never use IDs for styling.

We make sites Responsive by default as a Future Friendly measure.

We prefer to move into HTML, CSS, and JS sooner rather than later and build Front-end Style Guides (something like Pattern Lab) that evolve into pages and templates.

JavaScript

We write unobtrusive, js-hinted, JS. We keep all our JS in a few, minified, external JS files. We don't use inline script blocks or write inline JS. We only include jQuery when really necessary, preferring vanilla JavaScript code and micro-frameworks.

We Cut the Mustard to serve less capable browsers just the core, lighter and faster, experience, rather than send them lots of code they will struggle to run.

CHAPTER 11

Contributing back

Many of our components in github are open source. In the course of using them, you might find improvements are necessary or possible. We like having your contributions!

Please submit a pull request for our review. Although we don't recommend it, if you can't wait for our review and merge, you will need to fork that project on github and submit your changes to us as soon as the pressure is off. Please do create the pull request then.

CHAPTER 12

Production Deployments

Our DevOps team are responsible for all production deployments. This enables us to support the live sites and systems after hours, and ensure that infrastructural requirements like backups and monitoring are standardised.

Please note that production deployments need to be booked with the DevOps team by the appropriate Praekelt project manager, and that we deploy on Mondays through Thursdays.

CHAPTER 13

Things you need to know when starting a project

Sometimes there's a rush to get a project started. To spare yourself future trouble here's a checklist of things you need to have before starting any work.

1. You need to have been issued a purchase order.
2. You need to have been given a Scope of Work describing the thing you are to be building.
3. You need to have agreed to the timelines, estimates and deliverables described in the Scope of Work. If there are any ambiguities in any of the wording they need to be resolved before you start.
4. You need to have a clear picture of which stats need to be collected for the project and how those are to be stored to enable the people wanting access to those stats do the type of analysis they need to do. This differs per project so make sure you take the time to do this properly.

CHAPTER 14

Cheatsheet

If you want to get started as quickly as possible, here is a cheatsheet with working examples of what's required.

This will help you get up to speed with the way we do things fast.

The perfect dev setup

Our tools and software are centered very much around Open Source and Linux.

A lot of what we do happens on the command line.

In order to help you get started, we have provided setups for Ubuntu and Mac OSX.

If you are running Windows, we strongly recommend running an Ubuntu Linux virtual machine.

Ubuntu

1. Setup git:

```
aptitude install git
```

2. Get hubflow:

```
git clone https://github.com/datasift/gitflow
cd gitflow
sudo ./install.sh
```

3. Symlink git to hub:

```
sudo ln -s /usr/bin/git /usr/local/bin/hub
```

4. Test:

```
hub hf version
```

If you see this, or something similar, you are good to go:

```
1.5.2 - latest version
```

OSX Mavericks

1. Setup XCode Developer tools:

Download from the App store here: https://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12

2. Install homebrew:

From terminal:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

3. Get hubflow:

```
brew install hubflow
```

4. Test:

```
hub version
```

If you see this, or something similar, you are good to go:

```
git version 2.4.9 (Apple Git-60)
hub version 2.2.2
```

Working with Github

Cloning a repository

When working with our repostories, we would have created the repository for you.

Your next steps are to get it onto your local machine and initialize it for use with hubflow.

```
git clone
hub hf init
```

Example with that:

```
$ hub clone praekelt/ways-of-working
Cloning into 'ways-of-working'...
remote: Counting objects: 290, done.
remote: Total 290 (delta 0), reused 0 (delta 0), pack-reused 290
Receiving objects: 100% (290/290), 48.84 KiB | 71.00 KiB/s, done.
Resolving deltas: 100% (184/184), done.
Checking connectivity... done.
$ cd ways-of-working/
ways-of-working $ hub hf init
```

```
Using default branch names.

Which branch should be used for tracking production releases?
   - develop
Branch name for production releases: [master]

Which branch should be used for integration of the "next release"?
   - develop
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
$
```

This project is now ready for use with Praekelt's ways of working.

Writing code

Now that the repository is ready, you can now start adding code to it.

The steps are as follows:

1. Create an issue on github.

```
hub issue create
<enter text>
```

2. Start a new feature with hubflow named `issue-<issue # you created in step 1>-<description of work>`

```
hub hf feature start issue-1-going-to-write-some-code
```

3. Write code

This is where the actual magic happens.

4. Commit it

```
hub commit -a -m "hey look, real work!"
```

5. Push it back up to github

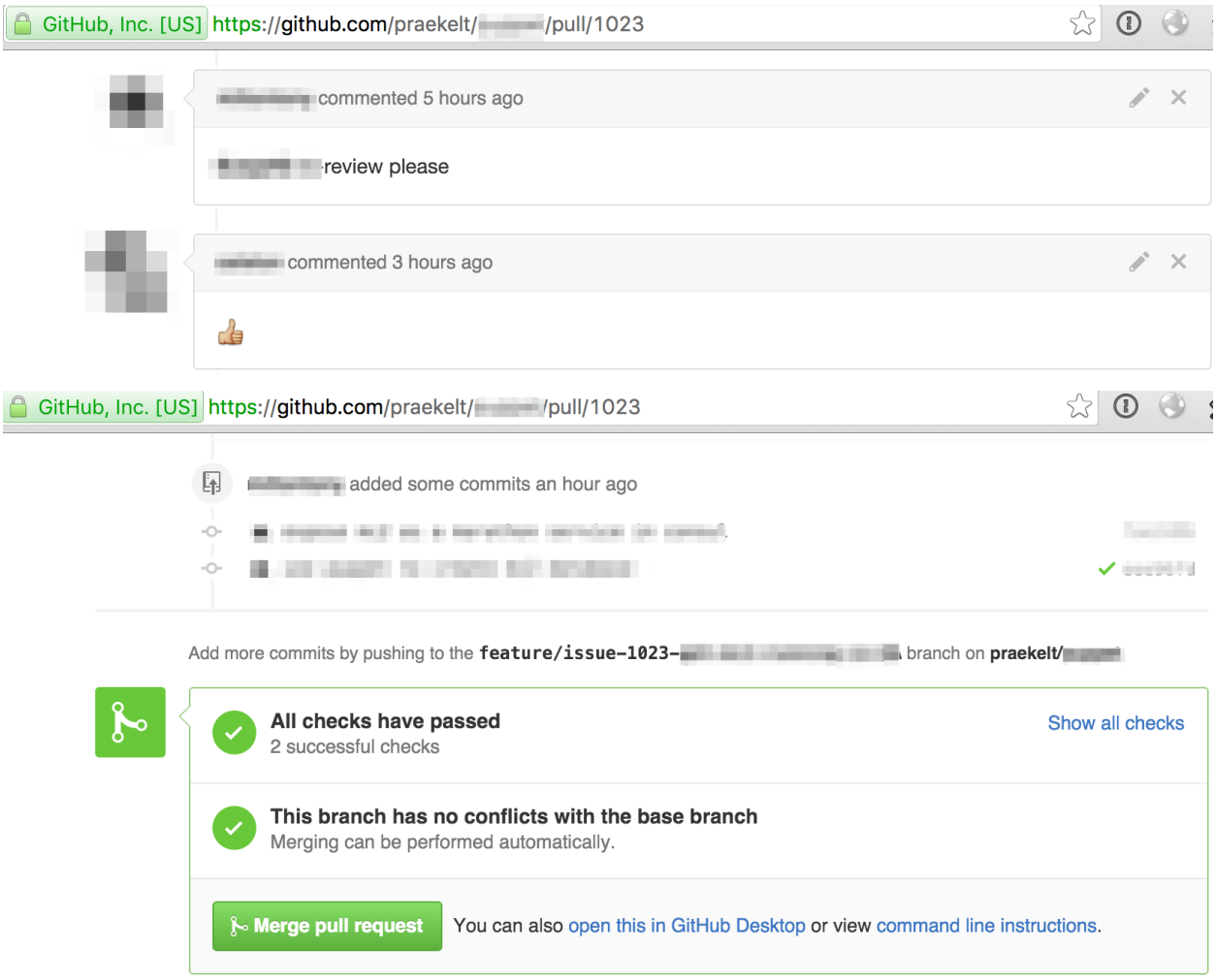
```
hub push
```

6. Open a pull request (PR)

```
hub pull-request -b develop
```

You can reference the issue by saying "Fixes #<issue number>" in the body of the PR. This will automatically close the issue when the PR is merged.

7. Get it tested (automatically #thanks-travis-ci), reviewed and +1'ed



8. Merge it into develop

9. Finish the feature

```
hub hf feature finish
```

10. Rinse and repeat

Merging develop back into your branch

Often your feature has "fallen behind" develop.

Before you can merge your code in you will have to merge develop into your branch.

Do this:

```
hub merge develop
hub merge push
```

This then merges develop into your feature branch and pushes it back to github.

Our coding best practices

We do this all the time, so here are a couple of 'quiet rules' we stick to:

- Write tests early on in the development process
- One change per feature (where possible)
- Always convert issues to pull requests (it just makes issue clean up easier)
- Commit often (smaller commits help in showing you what went wrong)
- When in need of help, generate a PR and ask for assistance
- Set yourself a deadline, if you haven't cracked the problem by your deadline, start talking to people