Praekelt Ways of Working Documentation

Release 1

Praekelt Foundation devs and individual contributors

December 02, 2016

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 follow and the things we would be expecting 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

Our Platforms

Praekelt uses two main platforms for the bulk of engineering work:

1. Django

   We use Django for websites, mobi sites, responsive sites, mobi HTML5 apps.

2. Vumi

   We use our Vumi platform for SMS, USSD and other messaging protocols.

Use of any other platform must be approved by our engineering management, and may need to be hosted separately from our usual environments, so please get this sorted out prior to commencing development on a project.

1.1 Django

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 accidently being deployed on a newer version without testing.

We strongly recommend you use our sample django-skeleton as a starting point for Django projects, as it has the layout and configurations we use, so that deployment will be smooth.

We deploy Django in the following stack:

- Ubuntu Server (current LTS release)
- haproxy for load balancing where appropriate
- nginx Web server that we use to run our Django applications in production.
- gunicorn is the application server we're using to service our Django applications with
- supervisord manages your processes and allows us to ensure that your applications are a) running and b) come up at boot should the server be restarted for whatever reason.
- postgresql for storing our data, usually for Django projects

For development, you can simplify this, and for QA we won't bother about haproxy but the rest of the stack will be required for QA so we recommend you keep your dev environment as close to this as you can.

Notes:

- We manage hostnames in nginx, because there may be multiple QA and live hostnames so don't use Django's ALLOWED_HOSTS.
- Make use of pip and virtualenv

- Avoid using CachedStaticFilesStorage, or generating CSS/JS automatically as this breaks load balanced environments.
- There are specific requirements for logging. Please see the *Logging* section for more information.

1.2 Vumi

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

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.

Apps can be written for Vumi Go, 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.

See the Vumi Go documentation as well as the JS Sandbox toolkit documentation for writing apps.

1.3 Logging

All logging must happen to a standardized path, under */var/praekelt/logs*. If your app has a large amount of files (celery, worker, and app, for instance), write each of these to a named path under */var/praekelt/logs/appname/foo.log*, otherwise just */var/praekelt/logs/foo.log* is sufficient.

Logs always end with the extension *.log*. *.err* is not valid. If you need to write out error logs, either use the name *foo-error.log*, or write your supervisord configuration with the *redirect_stderr* option.

The basis for this requirement is to ease debugging (hunting logs in 7 different directories is never fun, and causes issues when under time pressure), simplifies log rotation, and allows them to easily fit within our automatic log collection and indexing system.

CHAPTER 2

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.

2.1 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.

2.2 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.

2.3 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.

2.4 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

2.5 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.

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.

2.6 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.

2.7 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.

2.8 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.

2.9 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).

2.10 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.

2.11 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.

2.12 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.

2.13 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.

2.14 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 3

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 4

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 5

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.

5.1 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 6

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.

6.1 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.

6.2 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 7

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 8

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 9

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 10

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.

10.1 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.

10.1.1 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
```

10.1.2 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
```

10.2 Working with Github

10.2.1 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.

10.2.2 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!"
```

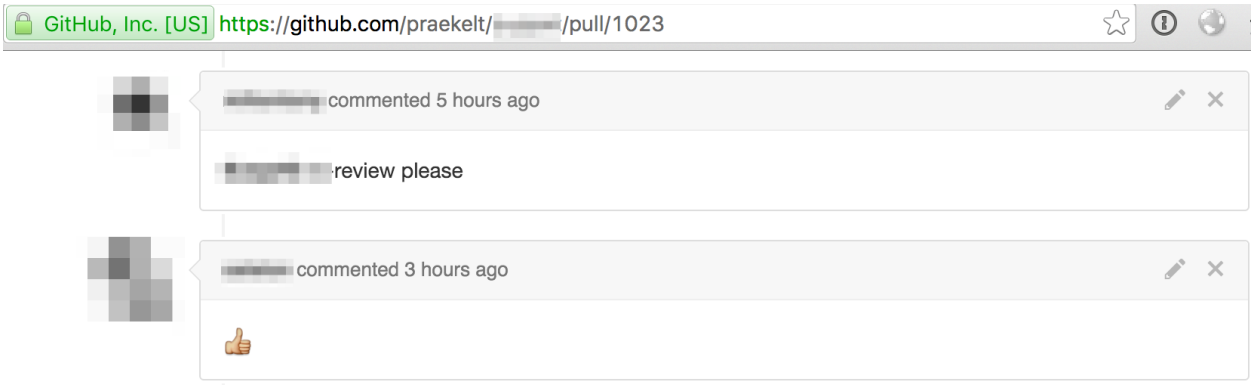
5. Push it back up to github

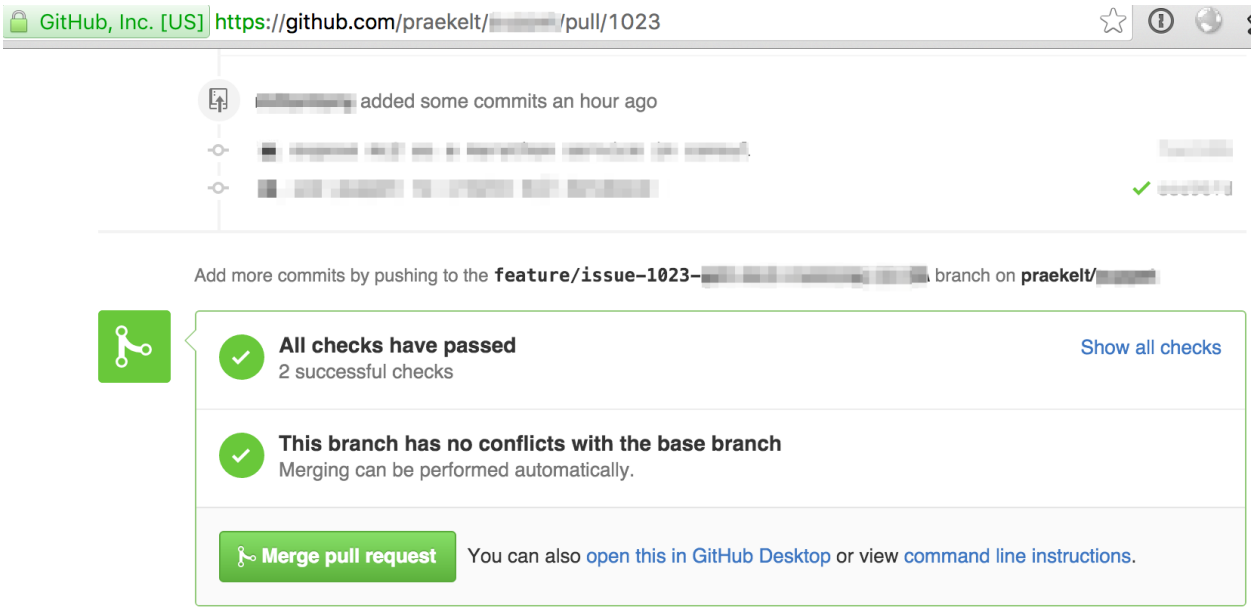
```
hub push
```

6. Convert the issue into a Pull Request

```
hub pull-request -b develop -i 1
```

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

10.2.3 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.

10.3 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