Praekelt Ways of Working Documentation

Release 1

Praekelt Foundation devs and individual contributors

Jul 31, 2018
1 Engineering teams 3
  1.1 Health ................................................................. 3
  1.2 Youth ................................................................. 3
  1.3 Site reliability engineering (SRE) .................................. 3

2 Projects 5
  2.1 Girl Effect Rwanda ............................................. 5
  2.2 HelloMama ........................................................... 6
  2.3 MomConnect ........................................................ 6
  2.4 Springster ............................................................ 6
  2.5 Tune Me ............................................................... 6

3 Infrastructure 7
  3.1 Nigeria ................................................................. 7
  3.2 South Africa .......................................................... 7

4 Tech 9
  4.1 CasePro ................................................................. 9
  4.2 Django ................................................................. 10
  4.3 Adding a new repository to Docker Hub ......................... 10
  4.4 HTTPS ................................................................. 10
  4.5 Junebug ................................................................. 12
  4.6 Molo ...................................................................... 12
  4.7 Python packages ................................................... 13
  4.8 RapidPro ............................................................... 13
  4.9 Seed stack ............................................................. 14
  4.10 Vumi ................................................................... 14

5 Tools we use 15
  5.1 IRC .................................................................. 15
  5.2 Git ................................................................. 15
  5.3 git-flow ............................................................ 15
  5.4 HubFlow .............................................................. 16
  5.5 Hub ................................................................. 16
  5.6 Issues & Tickets ................................................... 16
  5.7 Sentry .............................................................. 17
  5.8 Puppet .............................................................. 17
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Sideloader</td>
<td>17</td>
</tr>
<tr>
<td>5.10</td>
<td>Databases / data stores</td>
<td>17</td>
</tr>
<tr>
<td>5.11</td>
<td>Django Applications</td>
<td>18</td>
</tr>
<tr>
<td>5.12</td>
<td>Translations</td>
<td>18</td>
</tr>
<tr>
<td>5.13</td>
<td>Graphite</td>
<td>18</td>
</tr>
<tr>
<td>5.14</td>
<td>Front-end</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>Tutorials</td>
<td>19</td>
</tr>
<tr>
<td>6.1</td>
<td>Communicating</td>
<td>19</td>
</tr>
<tr>
<td>6.2</td>
<td>Tools that we use</td>
<td>20</td>
</tr>
<tr>
<td>6.3</td>
<td>Git and the basics</td>
<td>23</td>
</tr>
<tr>
<td>6.4</td>
<td>Gitflow and Workflow</td>
<td>24</td>
</tr>
<tr>
<td>6.5</td>
<td>Internship Overview</td>
<td>42</td>
</tr>
<tr>
<td>6.6</td>
<td>Creating Tagged Releases</td>
<td>44</td>
</tr>
<tr>
<td>6.7</td>
<td>Virtualenv and package control</td>
<td>48</td>
</tr>
<tr>
<td>6.8</td>
<td>Welcome Interns!</td>
<td>51</td>
</tr>
<tr>
<td>6.9</td>
<td>Workflow Cheatsheet</td>
<td>51</td>
</tr>
<tr>
<td>6.10</td>
<td>Your Machine</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>Intellectual Property</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>Our project process</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>Our development process</td>
<td>61</td>
</tr>
<tr>
<td>9.1</td>
<td>Example flow</td>
<td>62</td>
</tr>
<tr>
<td>10</td>
<td>Our front-end development process</td>
<td>63</td>
</tr>
<tr>
<td>10.1</td>
<td>CSS</td>
<td>63</td>
</tr>
<tr>
<td>10.2</td>
<td>JavaScript</td>
<td>63</td>
</tr>
<tr>
<td>11</td>
<td>Contributing back</td>
<td>65</td>
</tr>
<tr>
<td>12</td>
<td>Production Deployments</td>
<td>67</td>
</tr>
<tr>
<td>13</td>
<td>Things you need to know when starting a project</td>
<td>69</td>
</tr>
<tr>
<td>14</td>
<td>Cheatsheet</td>
<td>71</td>
</tr>
<tr>
<td>14.1</td>
<td>The perfect dev setup</td>
<td>71</td>
</tr>
<tr>
<td>14.2</td>
<td>Working with Github</td>
<td>72</td>
</tr>
<tr>
<td>14.3</td>
<td>Our coding best practices</td>
<td>75</td>
</tr>
</tbody>
</table>
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!
Engineering is split into 3 teams.

1.1 Health

The health team works on:
- MomConnect South Africa
- HelloMama Nigeria
- FamilyConnect Uganda
- NurseConnect South Africa

1.2 Youth

The youth team works on:
- Springster
- Tune Me
- The Internet of Good Things
- BabyCenter

1.3 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
2.1 Girl Effect Rwanda

Girl Effect Rwanda is a project to improve the lives of women and girls in Rwanda.

2.1.1 Publishing

The mobile site runs at http://www.ninyampinga.com/ and is hosted in South Africa.

2.1.2 Messaging

We use RapidPro and Junebug to interact with users.

RapidPro and Junebug run in 3 containers in South Africa:

- RapidPro web
- RapidPro celery
- Junebug

SMS

We use the shortcode 1019 to send and receive SMS messages.

We have Junebug config for both Mtech and MTN.

- Mtech is an aggregator for Tigo and Airtel users and we connect directly to them over SMPP
- MTN require that we establish an IPsec connection from vumi-gateway and then connect over SMPP


**USSD**

The USSD code is *900#*.

### 2.2 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`.

### 2.3 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`.

#### 2.3.1 MomConnect helpdesk

The MomConnect helpdesk runs CasePro and lets healthcare workers answer questions from the public.

### 2.4 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.

### 2.5 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

3.1 Nigeria

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

3.2 South Africa

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

3.2.1 Build

Build containers using Travis.
We used to use Sideloader but that’s deprecated.

3.2.2 Deployment

Deploy containers using Mission Control.

3.2.3 Monitoring
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.

4.1 CasePro

CasePro is an open-source Django application for helpdesk and case management. It’s made by the same organisation that builds RapidPro.

4.1.1 Pods

CasePro pods allow you to display custom data in the CasePro interface without merging it into the CasePro codebase. Currently you can only add information to the case detail view. Pods can be installed in the project as Python packages and added to the PODS array in settings.

**Registration pod**

Our registration pod retrieves user data from our seed microservices and displays it next to the open case.

**Subscription pod**

Our subscription pod displays information about the user’s subscriptions from our stage-based-messaging service.
4.2 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
```

4.3 Adding a new repository to Docker Hub

- Set up your own account on https://hub.docker.com/
- Get an existing admin (normally your team lead) to add you to the praekeltfoundation organisation
- Create a new repository under the organisation for your new image
- Add the automation team as a collaborator to your new repository with write access

Use the travis encrypt command locally to encrypt the praekeltorgdeploy Docker Hub password (from 1Password) - for example, encrypt REGISTRY_PASS=supersecretpassword. NOTE: if you are using travis-ci.ORG this will suffice. If you are using travis-ci.COM you may need to use the --pro flag e.g. travis encrypt --pro 'REGISTRY_PASS="secret"'. Travis CI are switching open source projects over to travis-ci.com and new projects will use .com.

Use the encrypted password environment variable to do docker login in your Travis file like this:

```
echo -n $REGISTRY_PASS | docker login -u "$REGISTRY_USER" --password-stdin
```

You can then push an image to Docker Hub using docker-ci-deploy:

```
docker-ci-deploy --version $VERSION --version-latest $IMAGE_NAME
```

4.4 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.

4.4.1 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.

### 4.4.2 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.

4.5 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.

4.6 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
4.7 Python packages

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

4.7.1 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

4.7.2 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.

4.7.3 Deployment

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

4.8 RapidPro

CasePro is an open-source Django application for messaging.

4.8.1 Tips

Adding credits

We don’t normally use credits because we host our own RapidPro instances and are billed by our SMS providers.

The permissions for adding credits are a bit strange. Administrators can’t add credits, you need to be a Django superuser. You can do this in a Python shell or in the database.

Once you’re a superuser, it’s `really` hard to find the interface for adding an arbitrary Top Up.

You can go to `/org/update/1/` and get a button to go to the Top Ups interface where you can add a new Top Up.

The maximum number of credits in a Top Up is the size of an `IntegerField` in Django, which is 2147483647.
4.9 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.

4.10 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.

4.10.1 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.

4.10.2 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.
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.

5.1 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.

5.2 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.

5.3 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.

## 5.4 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

## 5.5 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.

## 5.6 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.
5.7 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.

5.8 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.

5.9 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).

5.10 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.
5.11 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.
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.

5.12 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.

5.13 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.

5.14 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.
This section contains a deeper look at some of the technologies that we use and how we use them.

6.1 Communicating

Praekelt is a distributed company, which is a fancy way of saying the no-one is really obligated to be in the office when doing their work, as long as they're doing it. Interns are strongly encouraged to spend the majority of their time at the Cape Town or Joburg offices, at least initially, in order to get to grips with everything.

6.1.1 Slack

But wherever you are, communication is fairly critical as a software developer. So we use an awesome communication tool called Slack for the majority of our communications and for staying up to date with company/office news. We rarely use email and that’s usually reserved for things like contract discussions. If you already have a Slack account or have no idea what it is, don’t worry, you will simply need to supply one of our admins with your email address (preferably not one associated with another organization) and you will be sent an invite. Slack is a great multi-platform messaging system and you can get the integrated app for most platforms, although the browser version also works well. You’ll get the hang of using Slack pretty quickly, but check this out for more guidance.

6.1.2 Email

Email is not that important at Praekelt, but you may need a Google account, because we do use Google docs and Hangouts.

6.1.3 VoiP

We use Skype and Google Hangouts interchangeably. As an intern, you won’t use it much, but if you need to screen-share in a presentation, it would be a good idea to have a Google account and the Hangout plug-ins installed in your browser.
6.1.4 Sharing Documents

The easiest way to share documents is to drop and drag it into Slack, or share a Dropbox/Google Drive link in Slack.

6.1.5 Sharing Code

Any and all code MUST be shared using Git and GitHub, but this a fairly big topic that we’ll deal with in another article.

6.1.6 In Summary

Make sure that you have the following:

- A Google account
- Skype and Google Hangout Plugins installed
- A Github Account

6.2 Tools that we use

This article is going to briefly cover some of the tools you’ll need to work here and then go into some of the specific tools that we use and that you need to install. If you run into any trouble installing any of the necessary tools, ask us for help :)

6.2.1 Get familiar with a text editor

There’s a ton of debate about what’s better for a development environment in terms of text editors and IDEs. At Praekelt, almost everyone uses a text editor. Popular ones include VSCode, Sublime Text, Emacs, Vim and Atom. Sublime Text will give you an unlimited free trial if you’re happy clicking ‘ignore’ on sporadic pop ups asking you for cash. The others are free. A text editor is preferred because it gives you more flexibility with regards to editing various types of files and they all have some great add-ons (packages) that provide you with added functionality. Having said that, if you have an IDE that works for you, great :) As long as you’re using a tool that you’re comfortable with and enables you to get the job done, we don’t mind. If you’re at a complete loss as to what to get, I’d suggest VSCode.

6.2.2 pep8 Linter

There are a bunch of options out there, but make sure you get some kind of add on to check that your code adheres to pep8.

6.2.3 Homebrew

For those of you with macs, install Homebrew. Those of you with Ubuntu will already have this functionality with the apt-get command. This will make it much easier and faster for you to install software. Instead of downloading a file, unzipping it and manually running an install, this can all be taken care of from the command line. e.g:

```
brew install node
```

or for the Ubuntu crowd:
6.2.4 Gitflow and hub

It will make your life much easier if you install gitflow and hub, in order to adhere to our ways of working process.

For Mac:

```bash
$ brew install git-flow
$ brew install hub
```

For Ubuntu:

```bash
$ apt-get install git-flow
$ apt-get install hub
```

6.2.5 Virtualenv and Pip

You must install virtualenv. Basically, virtualenv allows you to have different versions of software and packages on your machine at the same time. See the virtualenv article on what it does and how to use it. You should have python installed on your machine, which has pip bundled with it in later versions. Pip is a package management system used to install and manage software packages written in Python. Just to make sure, do the following:

```bash
$ easy_install pip
```

Then use Pip to install virtualenv

```bash
$ pip install virtualenv
```

6.2.6 Tools that some of the team recommends

I took a brief poll of some of the devs for tools that they use and find useful. Here are a couple that you may want to look into.

- for code editors
  - emmet
  - gitgutter
  - jshint
  - various syntax highlighters
- for node
  - nvm for running different node versions
  - npm for package management
  - grunt, specifically grunt `watch` tasks for auto running tests
- jq, sed, and all the other command line things, (sort, uniq, cat, comm, etc)
6.2.7 Tools that I use

Here are some of the tools that I’ve found helpful. You’re under no obligation to use them and they’re mostly mac specific.

iTerm

 iTerm is a better looking version of Apple’s Terminal and has added functionality. It also works well with oh-my-zsh. (Mac only)

Oh-my-zsh

Oh-my-zsh is basically an add-on to terminal. It looks pretty, but can also give you helpful information while you work, such as your present working directory, what branch of a repo you’re in and whether your git repo is up to date. One of its coolest features though is tab completion, which is helpful when working with gitflow.

Skitch

Skitch is just a helpful image tool, especially for front-end work, for capturing screenshots, adding annotations and sharing with Slack. (Mac Only)

6.2.8 In Summary

We don’t expect you to have this all done when you arrive here. It’s likely that you’ll run into problems here and there or need help or are unsure about something. That’s ok, we’ll be there to help you out. This is just a checklist that you should check before starting actual coding work.

Make sure that you install/have the following

• A text editor/IDE with a pep8 linter installed
• Homebrew (if you’re on a mac)
• git-flow and hub
• Python
6.3 Git and the basics

Git is going to be a huge part of how you use and write software here. Lots of articles have already been written about git so we’re not going to repeat what’s already been said:

If you’re unsure about git, or need a refresher, check out this article to get the basics right.

Familiarize yourself with the syntax of git, because we will expect you to be able to do this in the terminal, even if you prefer a GUI. However, there are some helpful tools that make working with git easier. I use GitX Kraken, a multi-platform git GUI, to see the changes that I have made before committing. There are also tools for the terminal like oh-my-zsh which provide helpful information about git repositories within the command line (for Linux and OSX).

I speak from experience when I say that git is not always the easiest thing to get right and I’ve managed to bugger things up a fair amount. If in doubt, ask someone!

6.3.1 Github

You should have a Github account, if not, please get one. You need to send your Github account details to one of the admins at Praekelt, who will add you to our organizations; Praekelt Foundation and/or Praekelt (this may vary depending on what you’re working on). Almost all of our work is open source, so anyone can view and clone our repos. However you will need to get push permissions for each repo that you work on. Contact the appropriate admin. Familiarize yourself with pushing and pulling with Github. When you use `git clone` with GitHub, use the SSH version of the repo URL, not the HTTPS URL.

Helpful tip! Every time you push to Github, it asks you for your username and password which can get pretty annoying. There’s a secure way to skip this step, using SSH. Use this article to get it set up. Note that this links the machine that you do this on, to your Github account, so only do this on your machine.

6.3.2 In summary

• get to know the basic git commands
• make sure you have a Github account
• ensure that you get added to the Praekelt organizations on Github when you get here
• use SSH to speed up the git process on your machine.

6.4 Gitflow and Workflow

So how do we use git, Github and Gitflow to create our work flow and make good quality software?

6.4.1 A Broad Overview

In the development process, we need to think about branches of code and how they interact over time. In our process, there are 3 types of branches; Master, Develop and Feature branches. This structure is not enforced by git or Github, but it’s the approach taken by Gitflow.

So, our branches. Our ‘busiest’ branch is the develop branch. This is where the software product as a whole is developed, where things are tried out and the branch that all the devs are developing. When we’re finished with what we’ve been working on, we can then create a ‘release’ using Gitflow. This is done by taking everything we’ve done in develop and merging it in to the master branch, so that the master branch is always pointing at the latest ‘tagged release’. The master branch is the final product, it’s the software that the customer or user is allowed to use. In our case, it’s the software that is uploaded to the servers. Each merge in to the master branch has a release number (e.g. v0.0.2 or v1.0)

But it would be too complicated for all the devs to be working on different ideas at the same time on the same branch. So to do this, each dev or dev team works on a different branch, a feature branch. So, say for example we’re working on a website and we want to

• add search functionality
• display a picture along with articles
• fix a bug

Finishing those objectives will constitute a release. A feature branch will be created for each of the features (surprise!).

So, each branch gives a developer an isolated workspace in which to make changes and add code, without breaking the whole system.
The feature is finished once the changes have been reviewed and tested (a process explained later on in detail), we will merge that branch back into the develop branch.
This process will continue for each branch, until we’re finished with each feature and have merged it back into the develop branch.
And finally, once we’ve added the features we wanted, and we’re sure that our develop branch is working like its supposed to, we can create a release.
This is an overview of how the process works. In the next section, I’ll go more in depth about the commands you’ll use, and how you’ll be working with a local repo on your machine and communicating with the repos on the GitHub servers.

### 6.4.2 The Detailed Process

The first thing we need is a repository to work on. This will already be created on GitHub under the Praekelt or Unicore Organizations. You will the clone the repo using

```
$ git clone <REPO SSH URL>
```

So for example

```
$ git clone git@github.com:universalcore/springboard.git
```

Congrats! You’ve now got a local version of the code that’s hosted on the GitHub servers. It looks something like this:
Note that you only have the develop branch on your machine. This is because it will only clone the default branch from github. You need to set up a local version of the master branch and then use gitflow (make sure it’s installed) to take care of some of stuff behind the scenes, which you don’t need to worry about.

```
$ cd <repo name>
$ git checkout master
```

This will automatically create a local master branch and switch you to the master branch, switch back to the develop branch and use Gitflow to work its magic

```
$ git checkout develop
$ git flow init
```

You’ll get a bunch of options that you need to confirm that looks like this:

```
Branch name for production releases: [master]
Branch name for "next release" development: [develop]
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
```

Simply accept the defaults and stuff has happened!
Next, you need to create and start up your virtualenv, which is explained in a later article.

```bash
$ virtualenv ve
$ source ve/bin/activate
(ve)$
```

The (ve) that appears in the command line, is telling you that the virtualenv is activated. Then you’ll need to install the packages that will allow the code to run locally on your machine. The packages are listed in a file called `requirements.txt` in most cases and there may also be `requirements-dev.txt`. Luckily, we use pip to take care of all of that for us. Make sure your virtualenv is activated for the appropriate repo and then simply run the following commands:

```bash
(ve)$ pip install -r requirements.txt
(ve)$ pip install -r requirements-dev.txt
```

This is the standard way to set things up, but it may differ from repo to repo. Check out the repo’s readme file for instructions. There may also be a `bootstrap.sh` file in the repo, in which case you can skip the instructions from the virtualenv to here, and simply run:

```bash
$ sh bootstrap.sh
```

Great! So your local machine should be set up, but there are invariably problems the first couple of times. Ask a real live human to help you out :)

OK, so back to Github and Gitflow.

You’re going to want to create a branch to work on your feature. First, create an issue on GitHub, at the appropriate repo. This article explains how to do that. Once the issue has been created, GitHub automatically creates an issue.
number. It’s important because we use it to name our branches in a logical fashion.

Make sure your repo is up to date with `git pull` and now you need to come up with the name for your branch, which will look like

```
feature/issue--<ISSUE NUMBER>--<BRIEF DESCRIPTION, SEPARATED BY DASH "--" >
```

So you will need to enter this command

```
$ git flow feature start issue--<ISSUE NUMBER>--<BRIEF DESCRIPTION, SEPARATED BY DASH "--" >
```

So in our example we would use the following command:

```
$ git flow feature start issue-21-add-search
```

Now we’ve got something that looks like this:
You’ve just created a branch on your local machine, but you now need to show everyone what you’re doing by getting that branch on to the GitHub servers. You do this using the following:

```
git flow feature publish issue-<NAME>
```

So with our example, you would enter the following

```
git flow feature publish issue-21-add-search
```

There is now a remote version of your branch. You can also check on the github repo that your branch has been listed. If you get an error, the most likely cause is that you don’t have push permissions. Contact an overlord to give you access.
You’re now ready to actually start writing code, get cracking! As you complete each unit of work, you will create a commit. See this article about what’s in a good commit.
Now, you need to update the GitHub repo to ensure that if anyone wants to look at your code, it’s up to date. You do that using:

```
$ git push
```

and assuming no errors ...
That step is really important, along with `git pull`, if you’re working in a group and everyone is adding and changing things. Make sure you `git pull` and `git push` regularly and communicate with group members on what you’re doing, to avoid merge conflicts.

Once you’ve written your code, created passing tests and you think your code is ready to merge, you need to create a pull request. This allows other developers to review your code, spot bugs and suggest improvements using GitHub. This is an integral part of the code process. It allows developers to keep up to date, see what progress has been made and learn about new approaches or coding methods. Here’s an article about why it’s totes important.

Make sure you’re

- on the right branch
- everything is committed and pushed
- you have `hub` installed

then check the issue number again (this is why it’s important!) and then enter the following:

```
$ hub pull-request -b develop -i <ISSUE NUMBER>
```

In our example, it would be

```
$ hub pull-request -b develop -i 21
```

This will summarize all the changes you’ve made and will allow other developers to review your code. It will also test the entirety of your code using Travis and let you know if stuff is broken, but I’ll deal more in depth with testing in another article. Here’s an excerpt from a pull request (PR). Here’s an actual PR on GitHub.
Other devs are able to point out mistakes that can be made in your code.
You can always create a pull request before you’re completely ready, in order to make collaboration easier. When you want final approval, simply “ping” the appropriate superior using their GitHub handle or on Slack and ask for a review. If you’re on Slack, include the PR link. If they’re happy, they’ll give you a +1 or thumbs up, like so:

Just before you start celebrating however, you need to make sure that there haven’t been changes in the develop branch.
that conflict with the code you’ve crafted in your feature branch. It’s better to solve any potential issues before merging, so what we do is merge an up-to-date develop branch into our feature branch like so:

```
$ git checkout develop
$ git pull
$ git checkout feature/<BRANCH NAME>
$ git pull origin develop
```

You will then need to deal with any issues that arise and may need further review from devs.

Now, and only now, can you merge your feature into the develop branch using

```
$ git flow feature finish <BRANCH NAME>
```

In our example:

```
$ git flow feature finish issue-21-add-search
```

Note that the change is only made locally, SO MAKE SURE YOU PUSH! I have forgotten to do this last step before and ran into a multitude of merge conflicts when I finally realised my mistake :/

```
$ git push
```
So, develop now has a new feature! It will be up to a more senior developer to create the release using gitflow.

The local repository on your machine

So, develop now has a new feature! It will be up to a more senior developer to create the release using gitflow.
But remember that those changes will be made on the github server and not your machine, so `git pull` in order to get the updated version.

The local repository on your machine
6.4.3 Conclusion

There are some things that were not covered in depth here like testing and some of the tools we use, but this has been a breakdown and explanation of the workflow and how we use git, github and gitflow to maximize our code quality and make everyone’s jobs easier, although it might not seem very simple right now!

Things to note:

- Things like quality assurance from project managers and the broader software creation process is not covered here.
- Here’s an article that’s just about gitflow that explains it in a more general sense.

6.5 Internship Overview

6.5.1 Our Programming Language of Choice

Praekelt is primarily a Python shop. If you feel like you need a python refresher, I’d recommend doing a few online programming challenges to help remember some of the syntax structure. I use Coderbyte, but there are a ton of options out there.
6.5.2 OK, I can code, but how does the Internet work?

When I started as an intern at Praekelt, I knew how to program relatively small things like getting the palindromic primes between two numbers. It turns out that this is not particularly useful when you’re supposed to work on a website. Pretty much everything we do is connected in some way to the web and web technologies. If you’re familiar with things like frameworks, HTML, Javascript and CSS, that’s great. If not, that’s also OK. Like me, you’ll pick this stuff up as you go along. If there’s something you don’t understand, particularly in the first couple weeks, take your time and use Google, YouTube and other interns or devs to explain this stuff to you.

6.5.3 Code style

If a bunch of people are writing and collaborating on code, it helps if there’s a uniform style that everyone uses. We use pep8 for our python code. We’re pretty strict about it, in fact your builds will fail within the automated testing environment if you’ve failed to comply, like leaving trailing whitespace. This can be pretty frustrating, so get yourself an add-on to your text editor or IDE (like this or this) to check your code for mistakes or things that need to change. This takes the hassle out of learning all of the conventions and will make your life a lot easier.

You’ll also note that there are very few comments in our code. I was surprised to find this out after having been told time and time again that comments are reeeeally important. Basically, the approach here is that you should be naming your variables and writing your code in such a way that it is easy to understand and doesn’t need additional commentary. You’ll also notice that names/dates/headers, that kind of thing, are all left out of the code. Github already has that meta-information. We don’t need to repeat ourselves.

6.5.4 Testing

Our approach to development is that if you haven’t written tests for your code, you’re only half done. We test everything we write and as you get going with us, you’ll be expected to write tests that will test every part of your code. This means thinking about standard cases, edge cases and points of failure and recovery. The good news is that we have some excellent tools to help us with testing. You will run tests locally, but we also use tools like Travis and Coveralls so that every time you push code to Github, the code is tested all over again, showing you when a build fails, where it fails and if parts of your code are not tested. We compartmentalize our tests in what are called unit tests and with any changes you’ve made, Travis will run all of those tests again to make sure that stuff has not been broken with your changes. In fact, there is even an approach to coding where you write failing tests first and then write code to get the tests passing. You don’t have to do that, but it’s worth checking out. The specific testing packages that you use will depend on what you work on.

6.5.5 Some concepts

There are a couple of concepts/ideas/technologies/stuff that you should get comfortable with:

- How to use the terminal or command line, here’s a helpful resource for the command line, although it is centered toward Linux users.
- Git and version control, which is dealt with in the next article
- Web frameworks and what they are. I found this article and this video helpful. The web applications that we use are Django and Wagtail among others. Don’t worry, you’ll be given a lot of time to familiarize yourself with these technologies.

6.5.6 In summary

- We like Python
• There’s a fair bit of stuff to learn, but it’s OK, you’ll get there
• We adhere to a certain code format called pep8
• We test our code. A lot.
• We don’t expect you to know all of this stuff from the beginning

6.6 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.

6.6.1 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

Edit your app

Name

TuneMe Api QA

Description

(optional)

Docker image

prakelnfoundation/molo-tuneme:latest

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.

### 6.6.2 Warnings

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

### 6.6.3 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 Github.

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.

![TuneMe Api QA](image)

6.6.4 Conclusion

Congrats! We can now test our code in a QA environment before committing 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.

6.7 Virtualenv and package control

The purpose of a virtualenv and how it worked confused me at first, so I’m going to take a bit of time to explain what it is, what it does and how it works with pip.

Basically, virtualenv allows you to have different versions of software and packages on your machine at the same time. Instead of messing about with root folders and installs, you can use virtualenv to create an isolated environment in which you can use that version. So say for example, we want a local version of Django (a web framework written in python), but we don’t want a global version installed. (Assume Django is not installed on this machine) First we navigate to the directory we want to work in (in this case test), and create our virtualenv:

```
$ cd ~/test
$ virtualenv ve
```

The ve is the name we’ve given our virtualenv. This could be anything, but our convention is to use ve and naming it something else will cause issues with the Universalcore repos and probably most others as well. The examples below assume that this is what you have named it. After this command, you’ll get something like this

```
$ virtualenv ve
new python executable in ve/bin/python
Installing setuptools, pip...done.
```

If you run ls, you’ll see that there is now a directory called ve. Add ve to your .gitignore file if it’s not there already. This will avoid potential problems and git tomfoolery, like uploading all the install files to github. Ouch. The
The ve folder is where the settings are kept for this particular virtualenv, but it’s not currently “activated” or “on”. To do that we need to enter this:

```
$ source ve/bin/activate
(ve)$
```

See that (ve) before your command prompt? That indicates that the virtualenv called ve is active. So now we can install django. If we don’t specify a version, it will install the latest stable version of django. We’ll install v1.5.2 here for funsies.

```
(ve)$ pip install django==1.5.2
Downloading/unpacking django==1.5.2
  Downloading Django-1.5.2-py2.py3-none-any.whl (8.3MB): 8.3MB downloaded
Installing collected packages: django
Successfully installed django
Cleaning up...
(ve)$
```

Pretty easy, right? So now let’s check what we’re working with by opening python, importing django as a package, seeing what version we have and then quitting the python shell.

```
(ve)$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.VERSION
(1, 5, 2, 'final', 0)
>>> ^D
(ve)$
```

Now let’s “turn off” the virtualenv. This is done using

```
(ve)$ deactivate
$
```

See that the (ve) has disappeared? Now let’s try and use the django package:

```
$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named django
```

We get an import error because it’s not there; we isolated it within our virtualenv.

We create a virtualenv per repo that we work with. So we’ve been working in a repo or directory ~/test. Let’s create a second directory and install the latest version of Django. To help make a point, I’m going to name this virtualenv ve2. Remember that by convention we call it ve.

```
$ cd ~
$ mkdir test2
$ cd test2
$ virtualenv ve2
New python executable in ve2/bin/python
```

(continues on next page)
Installing setuptools, pip...done.
$ source ve2/bin/activate
(ve2)$ pip install django
Downloading/unpacking django
  Downloading Django-1.8.2-py2.py3-none-any.whl (6.2MB): 6.2MB downloaded
Installing collected packages: django
Successfully installed django
Cleaning up...
(ve2)$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.VERSION
(1, 8, 2, 'final', 0)
>>> ^D
(ve2)$

But what happens if I don’t deactivate the virtualenv and navigate back to the original directory called ~/test and use Django?

(ve2)$ cd ../test
(ve2)$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.VERSION
(1, 8, 2, 'final', 0)
>>> ^D
(ve2)$

See that we’re still working with the latest version of Django (that’s v1.8.2 not v1.5.2)? That’s because a virtualenv will work across directories. This may seem obvious because ve2 is still there, but remember that all our virtualenvs are called ve, so our helpful little tag (ve)$ is not going to help us remember to switch virtualenvs in our day to day work. So always remember that if you’re switching the repos that you’re working in, deactivate your current virtualenv, navigate to the directory that you want to work with (and that contains the appropriate ve directory) and then run the activate command again. The deactivate command can be run no matter what directory you’re in, but you must be in the directory with the virtualenv folder, in order to activate it.

(ve2)$ pwd
~/test
(ve2)$ deactivate
$ source ve/bin/activate
(ve)$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.VERSION
(1, 5, 2, 'final', 0)
>>> ^D
(ve2)$

Here’s a visual representation of how this all works.
Often when I’ve run into problems with running sites locally on my machine, it’s because my installed packages are not up to date. This usually helps:

```
(ve)$ pip install -r requirements.txt
(ve)$ pip install -r requirements-dev.txt
```

This will check what versions are installed and if any need updating, will update for you.

Next Article: Gitflow and Workflow

### 6.8 Welcome Interns!

Hi! Welcome to Praekelt.org

This section has been created to

- help explain to you how Praekelt.org operates
- what tools we use and why we use them
- how you can set up your machine correctly

You will be working with various people while at Praekelt, so feel free to ask them questions about the above, but also refer back to this site and linked sites. If you feel that there is missing information, mistakes have been made or you want to contribute to this site with your experiences and additional information, create an issue on the Github Repo.

### 6.9 Workflow Cheatsheet

#### 6.9.1 Setting Up Your Repo

Find the repo you want, get the SSH code
$ git clone <git@github:something>
$ cd <REPO>
$ git checkout master
$ git checkout develop
$ git flow init

You’ll be asked to set some settings. Accept the defaults (hit “Enter” key until it stops doing stuff). So it should look like this:

$ Branch name for production releases: [master]
$ Branch name for "next release" development: [develop]
$ Feature branches? [feature/]
$ Release branches? [release/]
$ Hotfix branches? [hotfix/]

The first time you enter the repo, you need to pip install some stuff.

$ virtualenv ve
$ source ve/bin/activate
(ve)$ pip install <something - check the readme file>

For the unicore repos, the pip install commands are like so:

(ve)$ pip install -r requirements.txt
(ve)$ pip install -r requirements-dev.txt

You’re set up :)

### 6.9.2 Using the repo

every time you enter your repo

$ source ve/bin/activate

### 6.9.3 Creating a New Feature Branch

Make sure your repo is up to date

`git checkout develop
git pull`

Create or find the relevant issue, get the issue number:

$ git flow feature start issue--<ISSUE NUMBER>---<BRIEF DESCRIPTION, SEPARATED BY DASH "-"->
$ git branch (just to check which branch you are on)
$ git flow feature publish issue--<ISSUE NUMBER>---<BRIEF DESCRIPTION, SEPARATED BY DASH "-"->

### 6.9.4 Creating a Pull Request

Make sure the relevant issue has been created within the repository, get the issue number that has been assigned and then:
6.9.5 Checking what other branches exist

This will show local and remote branches. Remote branches (branches on Github) begin with `origin`.

```
git branch -a
```

6.9.6 Merging a Branch

Make sure you've gotten a +1 or thumbs up in your pull request

```
(ve)$ git checkout develop
(ve)$ git pull
(ve)$ git checkout feature/<BRANCH NAME>
(ve)$ git pull origin develop
(ve)$ git flow feature finish <BRANCH NAME>
(ve)$ git push
```

6.9.7 Create a release

Check that you're on the develop branch

```
$ git pull
$ git checkout master
$ git pull
$ git checkout develop
$ git flow release start <NEW-VERSION-NUMBER>
```

You will now be on branch `release/<NEW-VERSION-NUMBER>` You then need to update the version number in the VERSION file using a text editor and then update the CHANGES.rst file, stating what has changed in the update.

```
$ git flow release finish <NEW-VERSION-NUMBER>
```

You will now be back on the develop branch

```
$ git push --tags
$ git push
$ git checkout master
$ git push
$ git checkout develop
```

6.9.8 Check your branch merge settings and details

Navigate to root of the repo

```
cd .git
nano config
```

Use whatever text editor you want, not nec `nano` to open `config`
6.9.9 To Delete a Local Branch

```
git branch -d <LOCAL BRANCH NAME>
```

6.9.10 To revert a commit

Navigate to the repo

```
git log
```

This will display you your last commits and their messages Find the uuid for the commit you want to undo

```
git revert <UUID>
```

Git will reverse the commit you made and apply that as a new commit

```
git push
```

and my personal favorite, but only to be used when you have messed up beyond comprehension …

6.9.11 When Things Have Gone Terribly Wrong

```
rm -rf <REPO NAME>
```

Then start again …

6.10 Your Machine

You should have your own laptop. If you don’t, let us know ASAP and we’ll see if we can sort something out.

We’re all about Unix-based machines, so if you have Linux or a MacBook already, that’s great. If you have a Windows machine and its your preferred OS then I’m afraid you’re going to have to do a bit of adjusting. There are three options in getting a Unix-based OS onto your machine.

For all of these options, you’re going to have to get an OS image. Since you’re reading this, I would recommend Ubuntu Desktop, specifically latest LTS release. As of the time of this article, that release is 14.04.2 LTS. You can get the image from someone at Praekelt, just ask around, or download it.

1. Fresh install of Ubuntu. This will wipe everything on your machine, but leave you with a nice clean install of Ubuntu. This is useful if you never want to see Windows again, or if you have a small drive. You can either burn the image to a DVD, or to a flash drive, or install from floppies. Click here for installation instructions. Select the “Replace Windows with Ubuntu” option when installing.

2. Dual boot with Windows. This will allow you to choose between Windows and Ubuntu when you start your laptop up. You do need enough free drive space for Ubuntu (at least 40GB free). Follow the same instructions as for (1), but select the “Install Ubuntu alongside Windows” option.

3. Run Ubuntu inside Windows in a VirtualMachine. For this we will use VirtualBox. Click here for instructions. You can also set up a headless version of Ubuntu, where you do all your coding in Windows, but execute the code in a headless (terminal only) VirtualMachine. This can be useful if you want to run a VirtualMachine, but your machine is low on resources. Instructions for how to do this can be found here.

The rest of these articles will expect that you are using Linux or OSX.
6.10.1 In summary:

Make sure

- you have a laptop you can bring to work
- that laptop is running Ubuntu or OSX
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.
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.
9.1 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-review, update, re-review ... +1 // ..
(ve)$ git flow feature finish issue-1-update-documentation
..// changes merged to develop by git flow // ..
(ve)$ git push
```
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 performance and accessibility audits.

10.1 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.

10.2 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.
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.
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.
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.
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.

### 14.1 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.

#### 14.1.1 Ubuntu

1. Setup git:

   ```bash
   aptitude install git
   ```

2. Get hubflow:

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

3. Symlink git to hub:

   ```bash
   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

### 14.1.2 OSX Mavericks

1. Setup XCode Developer tools:


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
```

### 14.2 Working with Github

#### 14.2.1 Cloning a repository

When working with our repositories, 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
```

(continues on next page)
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.

### 14.2.2 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. 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

### 14.2.3 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.
14.3 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