Bitmazk Pte. Ltd. Documentation

Release 0.1

Martin Brochhaus

November 03, 2014

Contents

Contents:

CHAPTER 1

Introduction

Contents:

1.1 Foreword

Bitmazk.

Beautifying bits and bytes.

We are a small web agency providing full stack services for Python and Django based websites and web applications. You give us your idea. We give you a working product.

For ourselves and when working on client projects we follow the lean startup approach which means that we try to get a working prototype out as quickly as possible and then iterate until the project reaches a point where all parties are happy.

We have a stable and constantly growing customer base and while our client work is currently our main source of income we are working on our own products in order to establish new income streams.

We are not funded and we want to keep it like that. Our main motivation to wake up in the morning and get some work done is the passion to create great products and the will to grow personally and professionally. It shall never be the urge to please some investors. Likewise the products that we create should solve meaningful problems and should not be money printing machines that exploit the weaknesses of people. You could say that we are not doing this for the money. We are doing this in order to create the work of our lifes'.

And one thing is for sure: This job _is_ life consuming. Almost everything that we do is the complete opposite of the ordinary 9 to 5 office job. We don't have offices. We don't have work hours. We don't have vacation days. We don't even have job descriptions.

How is this supposed to work? We are not quite sure, yet, but a key ingredient for our long term success is to create the happiest work environment in the world.

Luckily, as technology advances rapidly and even social and political systems evolve (not so rapidly) signs are good that the global virtual tech startup will be the standard form of employment in the near future. We want to be early adopters and take an active part in shaping that future.

This document is an ongoing effort to solve a few problems:

- To find out who we are why we do this and what we have to offer
- To describe every little detail about our company as a form of self reflection and self auditing
- To spread the word and find those rare crazy people that would love to work with us

To some extend we are hereby open sourcing our whole company. We think that we are good but we also know that we are not perfect. If you think that we suck please fork us on GitHub and change our company (or start your own).

1.2 Our Mission

If you aim for the impossible and reach just half of it, you are still going to amaze a whole lot of people. Therefore our mission is at the same time incredibly simple and seemingly incredibly hard to achieve:

We want to make the world a better place.

"Why do we want to do this?", you might ask. "Why would anyone NOT want to do this?", we would answer.

Without any doubt we are living in an age of technology. Machines are increasingly replacing humans, creating all kinds of problems:

- a global problem of unemployment
- an ever accelerating globalization
- an always increasing global energy consumption
- which leads to increasing exhaustion of natural resources
- and global warming

Our generation might not have it's World War II but that doesn't matter. We have global problems that might even be worse than that and while that sounds like a horribly negative depiction of our times it is in fact a good thing:

During times of great risk there are always also great opportunities.

While our age of technology created a whole lot of problems it did also create one incredible opportunity:

Knowledge is available almost for free, almost instantaneously and information can "go viral" and travel across the globe in a matter of hours. Ideas can be turned into prototypes without large amounts of investment capital or massive production sites. Think about Open Source. Think about startups that turn into billion dollar companies and render established corporations obsolete. Think about the Arab Spring. Think about Kickstarter. Think about Wikileaks.

In short: In human history there has never been a time where one individual had more power to make a global impact. This is the huge opportunity of our generation.

We believe that when given financial security and the possibility to make free decisions it is inherent in every human to do something and to do something good in particular.

At Bitmazk we are trying to create an environment where each member has the opportunity to grow as an individual, to live on this planet with their eyes open, to identify meaningful problems and to provide lasting solutions.

Ironically the ever accelerating technological revolution threatens humanity on many levels: environmental, physically, psychologically, ethically but at the same time it has raised the average standard of living so much that it allows each and every one of us to do the one thing that is truly human:

Making the world a better place.

1.3 Who are we?

We are self-taught. We are self-motivated. We are self-disciplined. We are self-aware.

1.3.1 Self-taught

All of us went through trainee programs in software houses and through university. In the end, all of us came to the same weird conclusion: Nothing that we have been taught at university has any relevance in our day to day work. The best way to learn programming is to have a mentor and to just do it.

The reason is not that our universities have been bad. The reason is that they usually teach two kinds of skill-sets: soft-skills and hard-skills. The soft-skills are kind of obvious. They are common sense and no one really needs to waste three years at university picking them up. You would learn how to negotiate in three days if you were forced to face a real customer in the real world anyways.

The hard-skills are in our case, of course, all related to programming and guess what: We already knew most of it *before* we started university. Most of us wrote their first BASIC programs at the age of 10 or so. After all that was the reason why we decided to study this field: We dreamt of turning our hobbies into our jobs.

Please take this with a grain of salt. If you wanted to invent the next programming language or worked as a software engineer for NASA, it would probably be a good idea to have some extensive university knowledge backing up your tasks. But we are doing web-development here. This is more like an art and less like a *serious* profession (it is still serious, but in a less life-threatening way than writing software for sending people to mars).

Our main skill-set is Python, Django, HTML, CSS, JavaScript. All of this is perfectly documented on the internet and backed up by huge and helpful communities. None of this is taught in universities or if it is, it is horribly outdated and of no use in the real world.

One thing is for sure: We socialized at university. We learned to cope with stress and deadlines. We learned how to think analytically. But the real skills that pay our bills right now are 100% self-taught.

1.3.2 Self-motivated

TODO: Write about how we need to know where we are going as individuals in order to fit into the company, wake up in the morning and push the company forwards even though no one is forcing us to do so.

1.3.3 Self-disciplined

TODO: Write about the threats of working from home and the incredible amount of self-discipline this demands.

1.3.4 Self-aware

TODO: Write how we are constantly questioning ourselves, trying to become more effective in our work without losing sight of our private life goals.

CHAPTER 2

Management

Contents:

2.1 Salary

Everyone at Bitmazk earns the same.

We don't know for how long we will be able to hold up this philosophy but as long as we have less than 50 employees we are quite sure that everyone's work will be equally important to the company's success.

This means that if you want to have a pay-rise, everyone will get a pay-rise and obviously that has to be earned, otherwise we would probably go bankrupt quickly.

We have a formula that determines how we handle pay-rises.

In simple words it goes like this: Every time we have one more month worth of total salaries in the bank, we get a S$100 pay-rise.

To be more precise, here are some variables:

- *starting salary* - the first salary when the company was founded
- *payrise count* - the number of pay-rises that have been given when we will get the next pay-rise
- *total salary* - equals *starting salary * number of employees*
- *payrise amount* - we set this to S$100

In order to get a pay-rise, the equity of the company (cash in the bank plus outstanding invoices) must equal:

```
(total salary + (payrise amount * payrise count * number of employees))) * payrise_count
```

This is at the beginning of the month, after all of last month's outstanding payables and salaries have been paid.

Example:

- Pay-rise count: *1*
- Starting salary: *S$4,200*
- Employees: *3*
- Total salary: *3 * S$4,200 = S$12,600*
- Necessary equity: *(S$12,600 + (S$100 * 1 * 3)) * 1 = S$12,900*

Employees	Salary	Total salary	Payrise count	Payrise amount	Necessary equity
3	4200	12600	1	100	12900
3	4300	12900	2	100	26400
3	4400	13200	3	100	40500
3	4500	13500	4	100	55200
3	4600	13800	5	100	70500
3	4700	14100	6	100	86400
3	4800	14400	7	100	102900
3	4900	14700	8	100	120000
3	5000	15000	9	100	137700

CHAPTER 3

Infrastructure

Contents:

3.1 Google Apps

- Connecting your domain to Google Apps
- Setting up Email
- Setting up Google Docs
- Setting up Google Calendar
- Setting up Rietveld for Codereviews

3.2 Telephone and Fax

- Setting up Sipgate
- Setting up Sipdroid on Android
- Using the online fax

3.3 Skype

- Using Skype for conference calls
- Using Skype for screen sharing (pair programming)

3.4 Time Tracking

We use [Freckle](http://letsfreckle.com) as our time tracking tool. It has resonable pricing tiers and is one of the most beautiful, intuitive and useful apps on the internet.

People

Every employee has one account. Freelancers can also be added as users, they are not able to see every project.

Projects

For every customer project we create a project in Freckle.

If one customer gives us several different projects, we would also create several different projects in Freckle. In this case we would group the projects alphabetically, by giving them similar names like `CN: Foobar` and `CN: Barfoo` (where `CN` is a shortcur for the customer name).

Additionally we add one project for our own company. When we have company meetings that are not related to one specific project, we will track the time here.

Finally every team member get's their own project as well. When you are doing some research about some new technology or working on a personal side-project and you feel that the work could be (some day?) relevant for the company, you can track your time here.

Tags

Every entry should have a tag. We try to keep the list of tags as short as possible. If you are not sure about the available tags, just have a look at the tags page on Freckle. If you think that we will need a new tag very often in the future, announce it to the team and describe the reason.

Currently, we have the following tags:

django

This is the core competence of our company. Our day to day web development is tracked under this tag. If the gros of your work was writing `.py` files, you should probably tag it as `#django`.

html-css

This is the second most important tag that we have. It covers anything that touches html, less, sass, css files.

js

JavaScript and jQuery related tasks will be tagged as such.

design

When doing work in Photoshop or just quickly creating vanilla Bootstrap markup for a prototype, it should be tracked as `#design`.

open-source

Sometimes we have to update old open-source apps or react to issues or pull-requests. This is usually not covered by any customer budget, so it should be tracked as `#open-source` on the internal company project.

documentation

As the name suggests, when time is spent writing documentation (usually user manuals for the customer), it will be tagged as such.

FREE*

Sometimes we are involved in a customer project but for some reason we don't want to tag the task as billable hours. In these cases, we add the `#FREE*` tag to the other tag that describes the task.

communication

Whenever we talk on IRC, Skype, Hangouts, phone, email or in person for an extended amount of time, we tag the time as communication and roughly describe what the conversation was about (just a few words).

maintenance

This covers simple tasks like adding content to the CMS. Usually those are tasks, that the customer could do themselves but for some reason prefers to delegate the task to us.

codereview

Like the name says. When we do codereviews for a colleague, we tag the time as such.

server

All tasks that are not real programming tasks but infrastructure tasks are tagged as `#server`. This includes setting up new Webfaction servers, adding cronjobs, running deployments, restarting the webserver and so on.

infrastructure

When setting up new subdomains or setting up a new server to install an internal tool (like Gitlab), this is not really programming work and it is also not server work (in the sense that we sell this work to a customer). Instead, this is internal infrastrucutre work and should be tagged as such.

setup

When you need to mess around with your local development environment in order to be able to start working, you can track this time with `#setup`.

research

When venturing into unknown territory where new technology needs to be used, a lot of time is usually spent with trial and error, googling, reading READMEs, setting up test-servers or local instances and so on. This time can be tagged as `#research`.

finances

Time spent on updating our internal finance tracking app or creating and sending invoices will be tagged as `#finances`.

acquisition

Time spent for answering emails to potential customers or spent for meetings with potential customers is tagges as `#acquisition`.

management

Anything that doesn't fit into any of the above tasks but somehow seems to be important to run the company an be tracked with `#management`.

3.5 Project Management

- Setting up Trello
- Layout of a typcal project board
- Conventions when using trello
- Tips & Tricks when using Trello

3.6 GitHub

- Our company profile can be found at https://github.com/bitmazk
- Even though we will add you to our team, you should watch all projects in that organization, otherwise you will not get email notifications when users create issues or send pull requests for our projects.

3.7 Barkeep

We use Barkeep as our main codereview tool.

The goal is that every single line anyone commits to any master branch in any of our projects gets reviewed by at least one other team member.

This might sound like a daunting task at first but it actually takes lesser time than one might think and creates huge benefits in the long run since the knowledge gets quickly spread from more experienced developers to less experienced ones.

3.7.1 Settings

After logging in for the first time you should go to `Settings` and

- set the line length indicator to 80 characters
- Set your name

3.7.2 Commits View

On this view you can created searches. You should at least have two searches:

- An empty search that shows all commits on all master branches
- A search for `branches:all` that shows all commits on all branches

Using the `Search options` for both searches you can set `Only show unapproved`.

With this setup you can easily see all unapproved commits accross all our projects.

Whenever you push your commits it will take a few minutes before Barkeep automatically fetches the commit. You don't need to do anything in order to upload a commit for review.

3.7.3 Workflow and rules in short

1. No one is allowed to approve their own commits.
2. If you feel unsure about the commit (maybe because you lack the experience) but you think that it looks good, do not approve the commit but leave a comment `LGTM` (Looks Good To Me). Someone else with enough experience will approve the commit.
3. If someone else already left a `LGTM` and you yourself feel unsure as well and you are the last employee to review, you leave a `LGTM` as well but you also approve the commit.
4. When someone leaves comments for you and suggests improvements, please do them in a new commit. The commit message should be `codereview fixes for 6jau2hrd`, followed by a more explanatory text of what you changed, so that we can still guess from the commit message what has been done, without needing to open the referred to commit in barkeep.

3.7.4 Committing

When the project is added to barkeep, you just push your code and after a few moments barkeep will have added the commit to the list. You can even push your feature branches to origin and they will show up in barkeep. Of course this has the disadvantage that all this code will show up again when you finally merge it. So far it has turned out that we rarely need to work on huge feature branches.

3.7.5 Issues

If someone finds something that needs improvement, he will comment on the line where the mistake is found. Also if something is unclear, questions will be asked in the same manner directly attached to the corresponding lines.

When you fixe an issue, reply with "DONE" to the comment that raised the issue. This way you know how many issues are left and the reviewer knows that a path with fixes is on its way. After fixing all issues add a general comment with "fixed in <commit number>" for reference purposes and to show, that the fixes have been committed.

If a fix is expected to be bigger, it is possible to add a Trello card for the issue and to link to it in a comment. The issue is then considered as about to be solved in later commits.

3.7.6 Suggestions

If there is something, that is not really a mistake but just a suggestion for improval you add "SUGGESTION" to the comment. Suggestions don't need to be fixed in the patch set like the regular issues.

3.7.7 Approving commits

If all issues are fixed or there were no mistakes at all, the commit can be approved.

If a reviewer is not 100% sure about the code, but has not found any real mistakes, he can add "LGTM" for "looks good to me". If the last one to review the commit would also add a "LGTM" he can just approve the commit, since no mistakes were found.

But keep in mind, that this can be a hint for the programmer that wrote the code, that it might need further commenting or better structure, since the code was not really understandable to all of his fellow colleagues.

3.7.8 How to add new repos

1. SSH into out barkeep server
2. Form there, SSH into the server that contains the repo. You will be asked to add that new server to barkeep's `known_hosts` file.
3. Add barkeep's public RSA key (`cat ~/.ssh/id_rsa.pub`) to the repo-server's `authorized_keys` file.
4. Now barkeep should be able to SSH into the server that contains the repo without needing a password.
5. Browse to our barkeep instance and go to `/admin/repos/`
6. Add the new repo (`username@username.webfactional.com:/home/username/webapps/git/repos/projec`
7. Reload the page. The new repo should be shown under the headline `Repos scheduled to be cloned`
8. Reload the page a few more times until the repo is no longer shown under that headline. Now scroll down all the way and check in the `clone_new_repo.log` if it says `Finished cloning the repo [reponame]`

CHAPTER 4

Workflows

Contents:

4.1 PyPi Release

We try to release as many components of our work as possible. As a result you can find a whole lot of our internal Django apps on Github at https://github.com/bitmazk

Once a package is mature enough, we also create PyPi releases so that anyone can easily install them via `pip install package-name`.

For the following description please understand the following terms:

Current release

This is the release that is currently the latest release on PyPi. The `CHANGELOG.txt` of a project could look like this:

```
==== (ongoing) ===

* Added this and that

=== 0.4 ===

* Added this
* Added that
```

In this example, `0.4` is the current release.

Ongoing release

Given the example `CHANGELOG.txt` from above, the ongoing release is labelled as `(ongoing)`. This is the release that you would get if you cloned the current master branch from github. You cannot install this release form PyPi, yet, because it is still ongoing and we don't know which version number it will get, yet.

4.1.1 Version numbers

It's a science of it's own to do proper version number management. Here is how we handle it:

- All packages start with version `0.1`.
- When we introduce a new feature, we increase the sub-version to `0.2`.
- When we add a bugfix or feature improvement, we increase the sub-sub-version to `0.2.1`.

- When we think that the package is finally feature complete, we increase the major version to `1.0`.
- A major version bump often means that the package might have backwards incompatible changes or that it depends on new major versions of other packages (such as Django).

4.1.2 Uploading a new PyPi Release

In order to upload a new PyPi Release for an existing package, the following needs to be done:

- Make sure that you are an `Owner` or `Maintainer` for that package, if not, ask the owner to add you as a maintainer.
- Develop your patch, always add a description of what you have done to `CHANGELOG.txt`
- Commit and push your patch
- Now it is time to prepare a new release. For this we need to bump the version number. Version number bumps should always be their own commits and never be done alongside normal patches.
- Open `CHANGELOG.txt` and add a new headline called `(ongoing)`.
- Remove the `ongoing` from the last version and give it a new version number. Above you can find more information no how we chose version numbers.
- Open `packagename/__init__.py` and increase the version number to the new current release.
- Commit those two files. The commit message should be: *Released vX.X*
- Run `git tag -a X.X` where `X.X` is the new current release.
- Add `vX.X` as a message for that tag.
- Push your changes.

At this point the new release is done on Github. The new tag signals to fellow developers that the master branch at this tag is stable and save to use. Now it is time to upload this release to PyPi:

- Remove the `dist` and `package.egg.info` folders
- Run `python setup.py sdist`
- Have a very close look at all the files. Make sure that there are no unwanted HTML files (i.e. from the coverage folder) or other unwanted files, such as `*.pyc` or `.ropeproject`. If this is the case, you need to improve the `MANIFEST.in` file, and repeat the last three steps.
- Also make sure that there are no missing files, such as HTML files of `templates` folder that you have newly added or `*.md` files for online documentation. By default the `MANIFEST.in` only includes all `*.py` files recursively below the package name. Anything else must be added explicitly.
- When the files included in the distribution look good, run `python setup.py sdist upload`
- Celebrate your new release!

Please note that at this point your projects of course do not yet profit from the new release. As we usually create new releases in order to fix a customers request, there is one last step left:

- Update `requirements.txt` files of projects that need the new release and deploy those projects.

CHAPTER 5

Code Conventions

Contents:

5.1 HTML

5.1.1 General HTML

Line length

Unfortunately it is not very easy to produce HTML that fits into lines of 80 characters. Therefore for HTML it is OK tp produce lines that are longer and aim for code that has very consequent indentation.

Quote marks

We use double quote marks (") to wrap HTML tag attributes and templatetag parameters.

Example:

```
<h1 id="anAttribute">Foobar</h1>
{% trans "This text is a templatetag parameter" %}
```

Indentation

Almost everything should be indented by 4 spaces. The only exception are `{% block %}` tags and `{% blocktrans %}` tags.

Example for a `{% block %}` tag:

```
{% block main %}
<body>
    <h1>First indentation</h1>
    {% for object in object_list %}
        {% if object.pk %}
            <p>{{ object.name }}</p>
        {% endif %}
    {% endfor %}
</body>
{% endblock %}
```

Example for a `{% blocktrans %}` tag:

```
<ul>
    <li>
        {% blocktrans %}
        Hello world! This is a blocktransified text. And we use it when we
        want to translate a big block of text, that possibly spans multiple
        lines.
        {% endblocktrans %}
    </li>
</ul>
```

Casing & class names

HTML tags and attributes are written in minor letters. CSS classes, names and IDs are written in as variables with dash in order to follow the naming conventions of the Twitter Bootstrap CSS framework.

Example:

```
<h1 id="unique-element" name="some-name" class="element-class"></h1>
```

Data attributes

Never reference IDs, names or classes in JavaScript. The risk that someone changes the class on an element and then accidentally breaks some JavaScript is too big.

If you need to identify a unique element via JavaScript, use *$('[data-id="element"]')* and give the element that attribute. If you need to identify a group of elements use *$('[data-class="elements"]')*. In fact you can use any attribute name in order to add specific settings that can be read by your JavaScript to all elements. We just prepend *data-* to all those attributes because Twitter Bootstrap does the same and because it is a good convention to indicate that this attribtue is used by some JavaScript.

Code blocks

Separate root level code via 2 empty lines.

Example:

```
{% block main %}
    ...
{% endblock %}


{% block extrajs %}
    ...
{% endblock %}
```

Ordering of attributes

ID, name and class are always the first attributes for a HTML tag. After that come data-attributes and then everything else. For input elements, the type shall come first.

Example:

```
<img id="id_foo" name="foo" class="foo-class" data-id="foobar" src="..." />
<button type="submit" name="btn-foo" ...>Submit</button>
```

5.1.2 Django templates

i18n

Always wrap all string in `{% trans "" %}` tags.

Example:

```
{% load i18n %}
{% trans "Hello World!" %}
```

URLs

Always construct all URLs with the `{% url "" %}` tag. Make sure to load url from future.

Example:

```
{% load url from future %}
<a href="{% url "object_delete" pk=object.pk %}">Delete</a>
```

5.2 JavaScript

We try to make our JavaScript code look as close as possible to Python code.

5.2.1 Line length

JavaScript allows to easily break lines, so we should try to keep our lines shorter than 80 characters.

5.2.2 Casing

Casing for variable names, classes, functions and constants should be the same as in Python.

5.2.3 Quote marks

Like in Python, we always use single quote marks (`'`), unless the string itself contains a single quote mark.

Example:

```
$(document).ready(function() {
    $('#someId').hide()
    var foo = "Let's do it."
});
```

5.2.4 Indentation

Like in HTML and in Python, we indent by four spaces:

```
function my_function(foo) {
    if (foo===1) {
        return 1;
    }
}
```

5.2.5 Lists

JavaScript has the ugly pitfall that the last item of a list cannot be followed by a comma. This can result in unnecessary bugs when someone inserts a new item after the last item and forgets to add the missing comma. Therefore we prepend the commas to the beginning of the list items:

```
var mylist = [
    item1
    ,item2
    ,item3
    ,item4
]
```

5.3 Python

5.3.1 Imports

- Imports appear in the following order:

  ```
  python builtins (os, sys)

  main frameworks (django)

  other frameworks / 3rd party apps (milkman, registration)

  our own stuff (appname.models)


  [actual code]
  ```

- Never import `*`
- When the import line gets too long, wrap it like so, again alphabetically:

  ```
  from foobar import (
      bar,
      foo,
  )
  ```

- Classes are listed alphabetically

5.3.2 Breaking lines

- Break long strings with brackets, each newline starts with a space:

```
('My super'
' long'
' string')
```

- Break long conditionals with brackets:

```
if (this == that
    and that == this):
    foobar()
```

5.3.3 Strings

- always use single quote marks (') and not double quote marks (")
- use double quote marks (") only when the string inside has a single quote mark ('):

```
bar = 'This is how we roll.'
foo = "It's just like this."
```

5.4 SASS

CHAPTER 6

Toolbelt

Contents:

6.1 Operating System

6.2 Autojump

6.3 Dotfiles Repository

- create folder username-dotfiles
- create README.md
- git status git add . git commit -am “Initial commit”
- login to GitHub
- create new repo “username-dotfiles”
- follow instructions for existing repo
- describe why git push -u
- add some dotfiles like .vimrc and .vim
- ln -s username-dotfiles/.vimrc and .vim

6.4 Git

- create .gitconfig

```
[color]
    diff = auto
    status = auto
    branch = auto
[alias]
    st = status
    ci = commit
    co = checkout
    br = branch -all
```

```
    log1 = log --pretty=oneline --abbrev-commit
    lg = log --graph --pretty=format:'%Cred%h..%Creset - %s %Cgreen(%cr)%Creset - %an' --abbrev-commi
[merge]
    tool = meld
[rerere]
    enabled = true
[core]
    editor = nano
    excludesfile = ~/.gitignore_global
[user]
    name = Prename Surname
    email = user@example.com
[http]
    sslverify = false
```

- .gitignore_global
- create your profile
- create your ssh key
- add your ssh key to your profile
- how to create a repo
- how to add a new file to the repo
- how to commit
- how to push (-u)
- how to remove a file from the repo
- how to add a submodule (init and update)
- how to update an existing submodule
- how to remove a submodule
- how to create feature branches
- how to rebase master into feature branches
- how to merge –no-ff feature branches into master
- how to delete history with gitk
- how to solve merge conflicts with meld

6.4.1 Workflow

- git init -> creates new git repo
- add some files
- git add .
- git commit -am “Initial commit”
- git co -b branch_name -> creates new feature branch
- implement new features
- git add .

- git commit -> add as many commits as you want

A new day dawns:

- git co master
- git pull -> get latest code from team members
- git co feature_branch_name
- git rebase master –> pull in code from team members into own branch and fix merge conflicts with `git mergetool`

When you are done with the feature:

- codereview.sh master -> like git diff master
- git co master
- git merge –no-ff feature_branch_name
- git st –> does everything look good?
- git br -D feature_branch_name –> a merged branch can immediately be deleted

6.5 SSH

6.6 Vim

- run vimtutor
- use dotfiles
- use pathogen
- install a plugin with pathogen
- setup vim as a python ide
- describe some useful movement keys to get around in the beginning
- how to add venv to .ropeproject * open vim in root directory of project * type :RopeOpenProject * vim .ropeproject/config * add python_path to ~/Envs/lib/python2.7/site-packages/

6.7 GNU Screen

- open new screen session screen -d -R name
- CTRL+A w –> show open tabs
- CTRL+A c –> open new tab
- CTRL+A :title foobar –> set tab title
- CTRL+A 0-9 –> jump between tabs
- CTRL+A :quit –> terminate session
- exit into shell –> exit screen tab, terminates session when exiting last tab
- CTRL+A ? –> show help

- CTRL+A :multiuser on –> enable multiuser mode
- screen -xr sessionname –> attach to running multiuser session

6.8 Regular expressions

- regexpal.com

6.9 Python

6.9.1 Virtualenv

6.9.2 iPython and ipdb

6.9.3 Fabric

6.9.4 Gorun

6.10 Django

6.10.1 Django documentation

- learn the tutorial
- beware: topics vs. references

6.10.2 Making queries

- always use pk instead of id (`objects.get(pk=5)`)

6.10.3 Test Driven Development (TDD)

General rules

- Whenever you create a file in a Django app, you also create a filename_tests.py file in the tests folder.
- If you are going to write an integration test (i.e. views_tests) then put it into a `integration_tests` subdirectory. That way we can chose to run unittests only and exclude integration tests for everyday work.
- Test cases are named after the thing that is tested. If we test a class named `Foo` then the test case will be named `FooTestCase`. If we test a method named `foo_bar()` then the test case will be named `FooBarTestCase`.
- Always add a docstring to each test case like so:

  ```
  Tests for the ``foo_bar()`` method.
  ```

  This makes sure that if we are testing functions, the test case name is not misleading (because it is CamelCase when the function was snake_case).

- Usually all tests for one class go into one TestCase. If you have a method that should be tested and that method is super complex, then you can create it's own TestCase only for that method (name it like ClassNameMethodNameTestCase).
- When you want to test different calls to a method with different parameters and that TestCase has a setUp / tearDown that does a lot of stuff on the database, it might me more performant to just write one test and call the method with all its different parameters in that one test. In this case please write assertion messages for each assert that describe which case you are testing (since you don't describe this through the test method name any more)
- In a TestCase class, the setUp and tearDown methods come first. After that getter methods follow, after that test methods follow. Test methods should appear in a logical order as you develop the app, simplest tests first, edge cases later.

Mixin

If your app has a model Foo and in your tests you need to add Foo objects to your database, then you should add tests/mixins.py to your app and write a FooMixin class that provides a create_foo() method. The method should also add the newly created object to the class, so that we can make assertions on it in the test without having to get it from the database again.

Models

- Before writing your model, test instantiation
- If the model has any methods that do stuff (for example get_full_name() or get_absolute_url()) then these methods need to be covered by UnitTests.
- We do not test behavior that is given by Django such as testing if blank=True works.

Views

- Before writing your views, test if the view is callable
- This can be done with self.client.get(url) or self.client.post(url, data)
- always use reverse() when referring to URLs
- Each ViewTestCase should have a method get_view_name() which returns the namespace:name of the view in this project.
- For views that need to be called with parameters, the test case should have a get_view_kwargs() method.

6.10.4 Fixtures

We make sure that a new developer can always clone the repository and run fab rebuild. As a result he must have a fully functional site with all test data needed to run all our selenium tests (that means, to test any possible thing that is possible on the project's website).

The workflow should be as follows: * Create your model * Add migrations for your model * Add an admin for your model * Login as admin, go the model admin, add some test data * Extend fab dumpdata task so that your new testdata gets dumped as well * All dumpdata commands should look like this:

```
dumpdata --indent=4 --natural appname > appname/fixtures/bootstrap.json
```

- run fab dumpdata and see if the generated .json file contains the model you just added via the Django admin.

- if it looks good, run fab loaddata. As a result, you should have the exact same data in your database as before.
- WARNING: Do not add tons of data in the admin, then extend fab dumpdata and then run fab loaddata. If dumpdata was faulty and you run loaddata, your db will be deleted and thanks to the faulty dumpdata command all your data might be lost. So: First enter just a few items, do the workflow, see if everything works, then add all the rest of the needed test data.
- Finally commit your changes like "Updated fixtures for appname". It is usually a good idea to make fixture changes as own commits and code changes as own commits.
- conventions: * user@example.com <– self describing usernames

    password: test123

  - admin <– admin user password: test123

6.10.5 Creating Models

- always add a verbose_name, almost always the same as the field name, just in a normal human readable form (without _ and stuff)
- we like to make CharFields of length 1,2,4,8,16,32,64,128,256... we dont't know why :)

6.10.6 South

- http://readthedocs.org/docs/south/en/latest/

Adding your new app to South: * ./manage.py syncdb, because convert_to_south assumes that models and DB are

in sync already

- ./manage.py convert_to_south appname

Adding new column to model * implement new code * ./manage.py schemamigration appname –auto * ./manage.py migrate

- how to add a column

CHAPTER 7

Continuous Learning

CHAPTER 8

Library

Contents:

8.1 RSS

8.2 Books

8.2.1 Self Growth

- Mindfulness in Plain English
- 7 Habits of Highly Effecitve People
- Stumbling on Happiness

8.2.2 Business

- Hackers & Painters

8.2.3 Design

- Design for Hackers

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*