Introduction

The frontend team headcount keeps growing, and each one of us brings new wisdom, new approaches, and new ways to think about everything we do. With each new person, parts of our codebase also become more and more disparate. Despite the growth, we want to maintain a homogeneous codebase, and a homogeneous product. We need to be roughly on the same page, so that wherever you go into the code, you'll feel at home, and wherever you go in the product, you won't be able to tell, just by looking at it, who made it.
With varying specificity, this handbook tries to cover all aspects of our everyday work, going from being part of the team to writing maintainable code, to creating good interfaces, to understanding the underlying data structures, and should give you a handle on how to think about what you do. The handbook is a perpetual work in progress, so please use the commenting functionality to suggest changes and additions.

1. Team

Frontend and backend together form product, and product, success, business, and operations together form the whole company. The roles of the teams are clean-cut, and competencies and responsibilities are easily determined. For the moment being, we are responsible for Dash - our analytics frontend.

Technically, our responsibility starts with data retrieval and processing from elasticsearch (through the python wrapper written by us), moves on to serving the django app using nginx+uwsgi (and every one of us should be comfortable with managing and monitoring the stack), and finally, the actual interface of our single page app in Angular (and everything it entails; html5 in short).

Skillswise, we are programmers, we are system and data analysts, we are user experience developers, and we are designers. The “user experience” part reminds that we are not merely creating screens - we are, more importantly, creating the flow of the user interaction - how it all connects, and what happens on any user input. What is their first impression, and what is the shape of the interface they interact with every day. We are not limited within the scope of the product, either - sometimes the solution to an interface problem is to change the culture that uses the product. It’s good to keep that in mind, too.

In some companies we would be the whole company. In other companies we would be called “full-stack”. However, the field we are in is vast, and so we are frontend - and the pond is as deep on the backend side as well (and each one of us should have at least a vague idea how deep it actually is).

As a team we are still small enough to make sure that everyone is on the same page - meaning, we should have a good understanding what others are up to, and could take over their work, if needs be.
We sync up by chatting in slack, via e-mails, and talking in google hangouts. The few formalized instances are the daily stand-ups and sit-downs, the weekly frontend hangout, and the internal frontend-only kick-offs.
When it comes to communicating to the company at large - we participate in the bi-weekly product hangout (demo what you want), and we become more active around the time we are ready to push a new version to beta.

Finally, to garner user input, apart from receiving feature requests and bug reports from success and the rest of the company, every now and then we schedule user interviews - goal of those is to get a feel of how a fraction of the user base is using our product by observing them in action.

Comms

Stand-ups and sit-downs

The stand-ups and sit-downs happen in the frontend channel. By writing a stand-up message, you signal everyone that you have started your workday, and a sit-down signals that you have just finished working. To write either, simply prefix your message with #standup or #sitdown, respectively.

Writing meaningful stand-ups and sit-downs takes time, so here are a few tips.

- The purpose of the updates is primarily to let people into your thought process, and the trickiest part is to find the right level of detail
- Before writing your stand-up, take some time planning out your day. It's ok to write your standup an hour or two into the day, once you gained the required clarity
- Similarly, reserve at least 15 minutes at the end of the day to write a meaningful sit-down
    - No matter the task, every single day has its own unique challenges. If the project you're working on takes 2 weeks, don't write "i'm still working on X" as that is known - instead, share the details of the specific day. And if it seems that the details would not be relevant, think how to explain them to everyone so that it makes sense. Imagine you spending 5 minutes telling your best friend about your day
    - If possible, try diving into your thought process - how did you end up with the specific solution and not some other one, what are your worries and do you have plans to address them, and so on
    - Don't be afraid to talk about problems and goals not met - by sharing your struggles, you are signaling to others that we all are still just human, and it might be that somebody else will have some useful suggestions for helping you
    - Also, lay out your plan for the next day to create a perpetual baton relay with yourself.

Weekly Frontend Hangout

We have a weekly frontend hangout to update each other on our progress in person, talk about any issues we are struggling with, and use the synchronous nature to garner some feedback. The hangout happens every tuesday, and includes more than just the members of

the frontend team. Most notably, both Andrew (CTO) and Mike (VP of product) are present. It's important to understand that during the hangout they are not fulfilling their respective roles, but are rather there because both of them happen to affect user interfaces through their jobs. The aim for the hangout is to share our knowledge and thinking, and to find good ways of expressing and receiving feedback. At the time of writing (may 2018) the hangout is relatively new in this specific form and the main goal for now is to simply learn how to talk to each other. The ultimate goal is to create an environment in which good feedback flourishes, creating a virtuous loop of sharing.

Internal kick-offs

For mid-size and bigger projects we start with a rough outline of requirements. Then one of us dives into the problem, and writes a more detailed solution. For this document, the idea is to try and play out all the combinations and permutations of user interaction in detail, to offer a full solution. This document then is read by everyone on the team and then presented and discussed in length during a kick-off meeting. At this point the solution is open for change, and the level of detail helps everyone to understand where this is going and how exactly it fits into the bigger picture. After the discussion and any adjustments to the detailed solution, the person moves on with implementing the first prototype. From there we start doing iterations. The motivation for these kick-offs is two fold: by sharing early and in detail we preempt any surprises and disagreement down the line, as well as we make it possible to talk about the project from now on as everyone now has a, within reason, decent understanding as to what will be done. It makes it possible to talk about the project in stand-ups, sit-downs and hangouts, without the need to dive into detail as to what this whole thing is about, and why we are doing it the way we are doing it. It also allows to play tag-team, when one party runs out of steam or needs a fresh pair of eyes.

Your strengths and weaknesses

Our field is wide and none of us individually can do all of it well. However, as a team we are whole. It's important to identify your strengths - as those are also your responsibilities - your competencies demand your participation in all things that you are good at. At the same time, by knowing your weaknesses, you'll know when you need more help than usual from your colleagues - using the help of others is encouraged. It's also important for all of us to know your ambitions - you are not expected to get good at everything, but knowing what you want to become stronger at will help your colleagues help you.

Everything belongs to everyone

This tends to happen quite naturally - you work on a piece of functionality, over time you become the domain expert, and then whenever anyone has any questions about that specific functionality, there is just one person to go to. That means, you don't get to go on vacations anymore, as nobody else can fill your role. That is exactly what we are trying to avoid. Nothing should belong to anyone specific, and everything should belong to everyone in general. For that reason, bugs are left unassigned until the actual work starts. And for that reason, whenever we do planning talks, the plans are kept vague on the "who".

Weekly updates and the internal kick-offs are here to facilitate sharing, but your attention also has to be present - make sure you have a good understanding about everything the team is working, as any new feature will be yours to maintain. What also helps, is picking up bugs that touch parts of the codebase you are unfamiliar with.

For the rest of the company we are a single unit of work. This does not always work in practice, but it is our stance, and you should speak up when you see this promise broken, as it has to be addressed. Ideally, all of us should be interchangeable - this makes for a more robust team, and this allows to avoid getting pigeonholed into one sequestered piece of functionality.

Collaboration entry points

You work in the team, and the team works within the larger context of the company. There will always be people to whom your work is relevant. As such, whenever you set out to start a new project, you have to plan for entry points for the others to participate.

We have formalized some of it: we have internal kick-offs that happen after you have familiarized with the problem and come up with a detailed plan. Similarly - the beta push is when we invite the whole company to participate. But these are just the outlines - you should actively plan out at which stages you'll open up for collaboration, comments and assistance. The effort will go a long way, reduce the risk of last-minute changes, and should generally make you feel better about the project. The mantra for this is "no surprises".

A good anti-pattern to watch out for here is the phrase "feel free to…" (let me know if you find something / if you come up with something / etc) - it sounds good and nice and polite, and most of the time people will nod along, but the onus now is on the other party that is also busy, and most likely they will feel free to ignore your feel free call.

Pairing up vs over-the-wall

Another antipattern we try to avoid is throwing things over the wall. In a typical example you might have a designer that has spelled out the interface to the pixel, and then the developer receives a mockup that they have to implement. Or, other way round - the developer is done, and they just tell the designer to “design it all up” now. This can happen in any situation where more than a single person is involved, and we end up with a game of broken telephone - the mockup isn’t implemented perfectly, the design has moved key interface elements to all the wrong spots, the backend data is not what we need, and the user actually needs something entirely different. The changes are intertwined - visual design and functionality go hand in hand, storing (backend) and reading (frontend) data are two sides of the same coin, and by working closely, you will get a glimpse of the thinking behind that other part of work, allowing everyone to iterate faster in an agile fashion.

And so the perfect antidote for over-the-wall is pairing up. Google hangouts with one party or even both parties sharing their screens works nicely for that. It can be a back-to-back speccing / designing / letter drafting sessions, it can be a mechanical turk, where one party gives instructions to the other and that creates discussion. It can also be sessions of pair programming. The process can stop once the outcome is clear. That then can be followed up by dropping offline to finish the task, and then coming back together to run through the next iteration. Sometimes pairing up seems wasteful, as only one of you is working. It also seems wasteful as it is just too much fun, and so much more easier. But in fact it’s much less wasteful than the alternative - you are saving time otherwise lost in back-and-forth, and you are saving yourself and everyone in the team from the potential frustration of pieces not fitting together.

Person over task

It might not be super easy to strike a balance here, but you should always strive to make yourself available whenever somebody needs your assistance, especially with people on your team. Same way, you can expect anyone not minding to chat with you when you need assistance, and you shouldn’t shy away from asking people to jump in a hangout with you - it’s part of their job! Jumping on a hangout can save hours if not days spent chatting, or understanding a particular piece of code. Also, a hangout is distinctive from a scheduled meeting, as it doesn’t carve out a buffer zone around it, it’s goal-oriented, and the headcount is limited, so there will be less passive staring at the screen. To put this one short - “busy” is a bad excuse.

2. Code

We are building a castle rather than a shanty town - additions and edits have to be up to snuff. The code additions have to be able to prove their worth, and they have to be written with readability-first in mind. Make it work, then make it right, then make it fast. It's easy to think in terms of one-offs, and just-a-little-addition, but that tends to quickly get out of hand.

You have to code strategically - as if this is the last thing you'll add to this code, and it will stay there forever for everyone else to read (and suffer). At the same time, avoid over-engineering. Even if we know exactly how the code could expand in the future, there is no point going there straight away. Winds tend to change quickly around here, and what seems to be within reach one month might turn out to be years away.

Don't be too clever, prefer explicit over implicit, use one-liners only if they are simple enough, and respect that not everyone will know as much about programming as you do, so keep things simple (as tempting as it might be to multiply a boolean with an arctangent to exploit a mathematical property). If it seems like the solution needs an explanation, either make the code more verbose or leave a comment (eg. if there is a silly business reason). And speaking of comments - keep an eye out for the non-obvious.

We are many and so it is imperative that our styles at least somewhat match up. Jumping into somebody else's written code should not trigger culture shock. One's handwriting shouldn't spill through everything they do. As such, be observant as to what style are we using, what style of code is the prevalent one, and then try adhering to that. Change is welcome, but we need to discuss it before implementing it. And don't worry - there is always time for a chat. For the basic formatting you can refer to our code style guidelines: https://github.com/Parsely/web/wiki/Code-style-guidelines#python.

And before you submit a pull request, take a step back to look at the whole corpus of the change, and consider if maybe it needs another cleanup and refactor first.

Fixing bugs

1. Understand the context
Being able to reproduce the bug is critical. For hard to reproduce exceptions when everything else fails, you might be able to walk through the code, compile it in your head, and understand how exactly the exception was triggered. Only once you have identified the bug and can describe it precisely, you can move forward.

2. Understand the history
Next you have to understand why is it happening - since most of our code has been running for a long time, more often than not the code before you will have a very strong reason to be exactly what it is. It could be that the original author hadn't anticipated a data case, or maybe the situation has changed; maybe the data is wrong - we have to be careful not to break existing functionality, and to do that we have to understand what the code does now, all permutations considered.

In other words - if you see an outright wrongness - dig deeper to understand how it came to be. Look into the code for comments and explanations, look at the commit history, and use the blame tab in github to figure out what exactly has transpired here.

3. Make a thorough fix and test it
Address the bug once you understand both, the issue, and the current code. And once you have fixed the bug, no matter how trivial, verify that the fix indeed works. Take time to consider any areas this code could affect (and don't forget mobile). Push to staging for testing, as sometimes things explode in the most miraculous ways.

4. Deploy to beta, and prod (when applicable)
The bug is not fixed until it has been pushed to beta and prod - once you have applied the fix, verified it, and considered any potential fallout, you should push to the appropriate target.

Dealing with a bugfix exploding in prod

Here's what you can do if a fix has ended up backfiring and prod is now exploding mightily:

1. If the problem and the fix is obvious - go for it, but don't rush.
2. If the problem is a bit more involved but you can still safely roll back (e.g. there weren't any db changes etc), check out the commit before yours, create a new branch off of that, and then push that to prod. This will give you time to fix the issue

3. If you can’t roll back, as it sometimes happens, slow down - prod being down is bad, but stressing out about it won’t fix it sooner either. Let the success slack channel know that you have killed the prod, ask them to put up a message, give a shout also in the frontend channel, and start fixing things.

3. Interface

Start with the Gnome human interface guidelines https://developer.gnome.org/hig/stable/. Not everything applies, but the guide contains many relevant and at times, arguably, timeless pointers. The thinking that is present in the Gnome HIG, permeates apple human interface guidelines, and matches up with the thinking behind Dash, as well.

The general tone of our interface is professional and calm. Distractions are kept to a minimum, and the focus at all times is on the data itself. We are read-heavy with just a few places where actual interaction comes in place (reports being the heaviest on input elements)

When thinking about the interface, always consider constraints - our users live in a world of distractions, interruptions, and under constant time constraints. We are simply not important enough to require full attention and careful observation from the user. We are "just another app", and what we want of them, is to remember us fondly for how we stay out of their way and help them do their jobs.

Keeping interruptions to a minimum

A good example for a necessary interruption that gives you what you ask for, is the login screen. For example, the user comes to dash from a link. Say, they want to see top posts for a specific day, filtered down by specific values. We detect that the user is not authenticated - so we show the login screen. But as soon as they log in, we forward them to the resource they originally asked for. While such behavior is expected from modern apps, it is not always obvious, and serves as a good illustration of what we aim at with dash.

A login screen is a severe case, as we can't show any useful information until the user really types in the credentials. This, however, is not so for rest of the interface. Just like Gnome HIG advises, we try to keep popups and modal interface to minimum. However, sometimes a popup is preferred over an in-place dialog to manage the complexity present on the screen.

Progressive reveal vs immediate actionability

Progressive vs immediate is one of the cardinal behavioral axes of interface design. Some examples of progressive reveal are:

- a wizard, where you are guided through several pages to provide all the necessary information, as opposed to dumping 50 fields into a single screentab pages - allowing you to split information up in more digestible chunks.
- an “Expand” link - show the most important information first and hide the less important parts behind an expander

Progressive reveal reduces cognitive load, and encourages the user to engage. Once a user has engaged, the cognitive cost has been paid, and the follow-up actions cost them less (in other words - taking the initiative is the most expensive part for the user). See Foot-in-the-door technique on wikipedia. Also, people are more likely to pay attention to information that they asked for.

Complexity can overwhelm and even alienate users, and progressive reveal is especially useful to counter that aspect as otherwise, once overwhelmed, they might decide to “not have time for this right now” and abandon the task entirely. An example to that was the previous iteration of reports - users felt that the reports look scary complicated to configure, and so refused to even try.

The trade-off of the progressive reveal, however, is fidgetiness, generally in the form of more clicking than what you’d require if you’d just dump all the elements onto a single screen. There is reason why at one point in the history of user interface development people started talking about “how many clicks are required to perform an action” - it’s a poor measure for gauging usability of the interface, but it does speak towards efficacy. So, ultimately you will have to strike a balance - how many steps does the wizard have, how do we reduce clickiness, is there a way to regroup items to offer a simpler view, and so on. But most importantly, it is important to keep both options in mind, and to understand their benefits and trade-offs.

Focus vs flexibility

Another way to look at the interfaces, is to consider the purpose of each screen.

A focused screen will have a clear-cut purpose that leaves no questions as to what the user is looking at and how they can interact with it. For example - “this is the page for author John Doe” - the context is clear, the expectations are clear as well - if you want to look at John Doe - open that page.

On the other hand, a flexible screen would be one that allows you to filter all data down to a specific author. The flexibility here is that you can change the author filter value at any moment, or even drop it entirely. The flexibility though, comes with increased complexity, but once the interface has been mastered, changing the author would take much less time for the user than in the focused example. However, the author now is just another way to filter - an apropos.

There is no better or worse here, and both interface approaches can be used in any situation. Keeping both approaches in mind and being conscious about the benefits and trade-offs, once again, will help you design good interfaces.

The top-down rule

Dash screens are laid out top-to-bottom, then left-to-right, and the element behavior follows the layout - interface elements that are lower in the page, can't affect data nor interface presented above them. The same goes for reading left-to-right between elements on the same vertical position.

Sane defaults and a healthy fear of preferences

Having a preference implies that all options are equally important.
As such "default sort order" is a sane preference as it depends on the focus of the publishing house and their monetisation system. "First day of week" is also a sane preference for historical reasons.

However, sometimes it's tempting to introduce a preference to avoid tackling a complex problem - the thinking roughly goes "we don't know so let's allow the user to decide". However, since we've been in the industry for so long, and analytics is what we do, we should know better. Instead of pushing the decision on the user and potentially allowing them to shoot themselves in the foot, we have to carefully weigh all the options of a proposed preference. Ideally, and not uncommonly, we are able to come up with a solution that works for everyone, without a preference. But, at the same time, we should be cognizant of situations where a preference is really required.

Graphing and presenting living data

A large chunk of data visualisation out in the wild focuses on static datasets - first you look at the data in its entirety, then you decide on the scope and the best way to represent it,

manicure the rough spots, and finally render a graph. In our case, however, we always work with living data, where numbers for the same chart will vary between a few hundreds, and a few billions. Moreso, the data patterns themselves differ between publishers. As such, when you look at the data, you should never focus on one set that will work really well, but rather consider the full spectrum - different sizes, different verticals, different referrer patterns, and consider how to deal with the range. This also means that there are many lessons from information visualisation that we can't apply without figuring out the variance first.
Some of the most important states that are easily overlooked are:

- No data
- Empty data (eg. all 0s)
- Outliers
- Spikey data (eg. the main timeline for small-volume sites)
- Mobile views

Superfluous precision

For bigger publishers, having total pageviews well into millions is pretty normal and so the final digits of such number represents a change of less than 0,01%. Such precision is superfluous and unactionable.

As a rule, our summaries are expressed in millions and thousands, but the listings will contain the full number. This is because in the listings it is too easy to miss the difference between a "1m" and "1k" label while scanning data vertically (comparing rows).

A few notes on maths

Avoid percentages of percentages

Sometimes we express values in percentages. For example "30% social traffic". If in the next interval the social traffic has risen to 32%, we might be tempted to express the difference as a percentage relative to the previous value (that is (32-30) / 30 * 100 = 6.7%). This is problematic because now we have a percentage of a percentage. To illustrate it better, consider the same growth, but going from 1% to 3% (3-1) / 3 * 100 = 66.7%. So growth in absolute terms was same, but now we have much more excitement.
A better way to deal with these is to actually subtract to determine the absolute change (2% growth). This way we represent the change in the original value, not the percentage. But getting one's head around the percentages of percentages is still not a very straightforward task, and as such we should generally stay away from this conundrum of multiple interpretability.

Switch to multipliers when percentages go over 100%

When a value goes from 3 to 300, that's a change of 10,000%. A "100x" in this case is a much more accessible figure for the users.