// DoctorC

Technical Design Philosophy

What this document about

This document lays down the high level approach of how we build software. This involves all aspects of software engineering such as -

- Writing Code
- Tests
- Deployments
- Production Management
- Software Infrastructure
- And more...

This document is meant to be the source of truth in how we operate from a technical implementation perspective.

If something we do in the “real world” that contradicts what is written here, there are only two solutions to it -

- Change this document to incorporate the “reality” way of doing things.

  OR

- Change the implementation to follow what is written here.

Hard to swallow thing - No software is the best software

We take the YAGNI principle seriously (more about YAGNI here). Don’t build something just because it's cool or to solve a problem that we expect to see in “future point”. We build software that **solves a really crucial pain point that is happening right now.**

When we write code, it has a cost associated with it -

- Cost of building it.
- Cost of not building something else because we spent time building this particular thing. It's called Opportunity Cost. This is the most important factor in prioritizing tasks that the engineering team is working on.
- Cost of maintaining (bug fixes and on call issues).
- Cost of upgrading and scaling it.

All these costs add up if we are not careful keeping it in mind.

It is important to ask yourself the question - “Do we really need to build software to solve this problem?”. Frequently the answer is “No” and we can solve the problem by -

- Not solving the problem. Let it happen.
- Creating a manual process.
- Using some off the shelf tool that is built to solve the problem.

If we can solve a problem without writing a single line of code. That is the best solution.

How to think about writing code

Principle - Code should be easy to read

This is one of THE most important principles we expect all engineers to follow. Code is a piece of logic that will be working for years and decades. So it is very crucial that you write code for reading. This means that how you structure your classes, functions and layers of abstraction should be well thought out.

Imagine someone coming to read your code 2 years after it has been written. The code style should be optimized to make it easy to read. This means that you have to be very explicit in choosing -

- **Correct NAMES for functions and variables -** This is one of the most crucial fundamentals that every engineer joining the organization will learn. We are anal about naming things. It is very hard to name things correctly in a readable manner. Some common mistakes -
    - Using short forms (num_ord) instead of use full forms (num_orders)
    - Non descriptive names (update_order) instead of descriptive names (update_coupons_in_order)
    - Reusing same/similar names for different things - OrderMiddleware which is not actually “django middleware” but just a regular class.
    - Using SomethingManager design pattern resulting in vague name and overloading a lot of unrelated functionality in the same place - more about it here. All because the name wasn’t thought of correctly.

More about writing readable code is written here.

- https://carlalexander.ca/importance-naming-programming/
- https://blog.codinghorror.com/learn-to-read-the-source-luke/
- https://blog.codinghorror.com/the-noble-art-of-maintenance-programming/

Principle - Code should be easy to extend

One of our core advantages over competition is our ability to move fast. We think hours, not in days. We want to ship changes to users as fast as possible. To support such a speed, we optimize for ease of development over other things like proper structuring, full test coverage, perfecting every piece of code, scalability via use of microservices etc etc

Ease of development is weaved throughout our software development process with things like automated tests, fast deploys, less bureaucracy etc.

It is also encapsulated in our principle of writing simple code but in a way so that it is easy to extend. For example - when we wanted to start importing reports from our partner labs' systems, we built our db models only for that partner lab in the beginning, but we kept the code flexible enough that when the time came to add a new integration, the changes to do those were not painful. In this example, the lesson is not to build generic systems from the beginning. It means that we write code in such a way that if we want to transition specific to generic systems, we can do so with minimal pain.

Principle - Understand when to optimize for speed of execution

Only write optimized code when the scalability problem is obvious and imminent. Don't try to write optimized code for a hypothetical situation happening 6 months down the line. We have not had any real scaling related problems until the past 6 months. Our approach to making this fast is to use more powerful amazon servers. They cost nothing compared to the engineering time that would be spent on optimizing every single inefficient problem in our system.

Also, don't forget that iterating over tens of thousands of data points is trivial for today's computers on a very basic level. Please think of the scale of what you are dealing with before deciding to write "fully optimized highly scalable" code.

Here is a link to performance numbers every software engineer should know.

Our approach to testing and QA

We write tests to check the quality of software. We **do not** write software to be centered around tests. We do not follow Test Driven Development (TDD) to the letter.

We treat tests as first class citizens in our software development cycle. This means -

- Each developer is responsible for writing the tests for the module they are developing/maintaining.
- Tests are held to the same standards as regular code. So all the principles mentioned in this document are applicable in writing tests too.
- We lean heavily on automated tests to catch regressions.
- We write comprehensive tests so every edge case is handled. Especially in critical flows related to booking appointments and payments.
- Breaking tests are bad and they should be fixed before production releases.
- End to end tests are totally worth it. You should spend extra time learning selenium and how to write selenium tests.
- Flaky tests are worse than no tests. Flaky tests should make you stop whatever you are doing and investigate the root cause and fix it. Flaky tests make the entire testing framework non trustworthy, so it is important to avoid them at all costs.
- We do not plan to have a separate team for QA. Having a separate QA team makes software developers ignore quality since it becomes “someone else’s problem”.
- Tests should be fast, we will continuously work on our infrastructure to enable our entire test suite to run in less than 5 mins. This is currently not the case, but we want to enable this.

Principles of Software Infrastructure

We like to keep the software infrastructure simple to use. It might be complex under the hood but it should be easy to use. Think of it like a car - driving a car is easy, but it is crazy complex underneath. We strive to keep the usage simple, even if it means that we go to unreasonable lengths to achieve that simplicity.

An example of “simple to use but complex underneath” infrastructure is our development environment or our home service APIs.

Principles of Software Security

Being in healthcare, security of software is a non-negotiable aspect of our development. We follow the BeyondCorp Security model that is developed by Google. This means that -

- We used Role Based Identity Access management systems everywhere. This is the whole “Users, Groups, Permissions” model.
- We follow the principle of least privilege in giving access to people to the various systems.
- We use 1password for all passwords (no exceptions).
- We use Secrets Manager to store all secrets (no exceptions).
- We try to use Google SSO (Single Sign On) as much as possible to avoid creating more accounts per system.
- We build “layers” of security, so in case there is a breach in any one part of our system, everything else is not compromised all at once.
- We use Multi Factor Auth (MFA) everywhere.
- We separate our work devices from personal devices, especially those for accessing production systems.
- We DON'T use VPNs as a blanket protection for things. It is too big a single point of failure for things.

A word on “ease of use” when it comes to security

Security inherently comes with the tradeoff with respect to “ease of use” of systems. The more secure the system, the harder it gets to use it. We try to balance as much as we can to make the security a non hassle part of our workflow. We also try to minimize the amount of things a human needs to do to keep systems secure, it should be automated as much as possible.

Deploying and Managing Production

- Deploying to production should be easy. The definition of “easy” is as follows -
    - If someone wants to make a change to some text on any page as fast as possible. How fast can we deploy the change? The ideal answer is less than 5 minutes. We are currently at less than 5 minutes (assume skipping of tests). We will always make sure to keep this number to less than 5 minutes.
- This way we can rollout fixes as soon as required as well as roll back to older versions of code too.
- As our Production Environment becomes more complex we need to add “observability” to the various components of the systems so that it's easy to figure out what's going on. This includes investing in things like distributed logging, distributed tracing, retrieval of logs etc.

// FINITO - GOODBYE