About the Team

I served as one of three full-time UX Engineers on a small yet dynamic team at Google. We were responsible for managing and maintaining all off-app websites for YouTube. Our team consisted of experienced senior level frontend engineers, and we would delegate project management, consulting, and engineering tasks to one member per project.

Rationale

With millions of users visiting YouTube daily, and the lack of a “YouTube About” site, there were bound to be many inquiries regarding its functionality and policies. To address these concerns, our team got involved and spearheaded the engineering efforts for the How YouTube Works project. It was aimed at providing insights into the platform's operations and clarifying what YouTube is — and is not doing — to foster a responsible community for its users, creators, and artists.

Anticipated Issues

I quickly figured out that the most difficult part of How YouTube Works would not necessarily be the coding, but rather the scalability, since this would be a “dumping ground” of information YouTube wanted to get out to the public. It had the potential to grow to an unmanageable size, especially with localization and new stakeholders coming and going at any given time.

Success Criteria

How YouTube Works is the sort of site that requires messaging to be on point and on time, so streamlining the process from making updates, adding new pages, to localization was crucial. The idea was that YouTube would have a way to respond to something quickly — COVID-19 for example. The stakeholders could add content to How YouTube Works with little oversight from an engineering team. This site was not meant to increase conversions, but rather to inform and to build user trust. The success criteria was:

1. Does it make it easy for users to find the information they are looking for?

2. Can stakeholders create, read, update and/or delete (CRUD) content on the site for quick responses?

In those two regards, I believe we succeeded.

Timeline

My rule of thumb for a frontend engineering timeline is two days per page, depending on the complexity. This is based on experience, so much so, that I was able to create a spreadsheet that took the number of pages, assigned a value based on the complexity for each page, and gave a good estimate of how many business days it would take to build the project from design handoff to launch.

Now, the engineering of each page is not the only task to account for. Feedback from stakeholders, QA, accessibility, localization, time to resolve any feedback, and some buffer time were also included in the tally. Too many times had we been given a project that was in design for months but only given a few weeks to develop. This timeline template spreadsheet gave the stakeholders a clear understanding of how long it may take… usually eight to ten weeks or more. Of course, there would be compromises and tasks done in tangent to help facilitate and shorten the timeline when necessary.

For How YouTube Works, using a templated design approach and launching in English only while content was being localized helped to drastically shorten the timeline.

Key Decisions

Most of the stakeholders lacked experience in building websites, especially for Google, which follows a particular process. Our team was generally the first one they interacted with before anything was put to paper, therefore, a lot of the key decisions came down to our expertise. For How YouTube Works, I was shown rough wireframes, and it was apparent that the stakeholders did not anticipate the same issues as me, particularly with scalability. Each page was unique and who knows how many would be added in the future.

I knew that the answer would be a design solution as well as an engineering one. I made sure I was involved in every design review with the agency and worked with them to come up with a solution that utilized templated designs, with some custom variations. This whittled the unique pages from dozens to just three or four. This allowed new content to be added and/or maintained quickly and drastically shortened the dev timeline.

Cross Functional

The stakeholders involved in How YouTube Works had no idea how to launch a website. We were the experts, so we took it upon ourselves to not only be the engineers but teachers. We wrote pages of documentation and walked them through getting approvals from privacy and security to legal reviews. We took part in every design review to consult and give our general feedback and ideas. From ideation through implementation and maintenance we would work with everyone: project managers, agencies, UX researchers, privacy experts, SEO experts, vendors, and even lawyers.

The process could be like herding cats. To help wrangle everything, we tried to minimize the number of communication channels. As much as possible, we kept all communication in a single Technical Design Doc (TDD). It included a sitemap, links to Figma files and other docs, locales, timeline, etc... All information was kept in one place to ensure everyone was on the same page.

Execution

Scalability was the name of the game for How YouTube Works. It was imperative that a new page or subpage could easily be added or removed, localized and launched really quickly without worrying whether it would break the site or not. ***Otherwise, YouTube risked getting bad publicity or even sued.***

The tools I had were an inhouse site builder written in Python and Flask and a CMS. Normally, we would hard code the paths in the backend. In this case, I wrote code in the backend to fetch documents from our CMS that would dynamically construct the paths. In this way and with a little training, any non-technical stakeholder could theoretically go into the CMS and create, read, update,

or delete any page on the site. They could then file a ticket to have the site built and deployed into production. And because we had vendors in India to help maintain How YouTube Works, an update could be deployed 24hrs a day.

Dynamic page creation was not the only novel solution that was needed. The pages were derived from templates, but the content was more or less like a blog. The stakeholder needed to be able to move elements, such as images, videos, paragraphs, around without having to re-localize everything or create schemas that covered every possible order of elements, which would have been impossible. To solve this problem, I developed a plugin for the CMS that allowed the stakeholders to write custom markdown in a textfield. The custom markdown would pull in other documents from collections that were composed of individual videos, images, buttons, lists, and other elements. It looked something like this:

```
## Headline 2

This is a paragraph with text blah, blah, blah.
And here is an element added with custom markdown:

[ @collectionID=documentID ]

1. List item 1
1. List item 2
1. List item 3
```

Impact

The success criteria for How YouTube Works was to inform users in a timely manner. I believe we succeeded in this. For example, stakeholders were able to put together this COVID-19 page to inform

users what YouTube's response has been as well as a page on supporting black communities. All done without any engineers helps, except for building and deploying.

Traps

The How YouTube Works project quickly became bloated with the number of pages and dozens of locales each page was localized into. This made maintaining the site tedious because building the site for production would take a long time.

I was naive to think that stakeholders would stick to the templated designs for each subpage. Stakeholders with different priorities saw the success of How YouTube Works and started coming to us with concepts like US of YouTube and the Creator Economy that derive from no templates. At that point, I was building sites within a site, further bloating the project with disparate code.

Lessons

After falling into this trap of adding more and more disparate code into one project's repo, we started breaking up How YouTube Works into separate repos and created a header component specific to this site in our library. That way if we needed to add or remove a page, it would reflect in the header for all projects in the new repos.

A Different Approach

The sheer amount of content on How YouTube Works would have been better handled outside of the CMS. The schemas got very complicated due to all of the localizations and locale-specific content. It required us to write docs and provide training in order to get stakeholders comfortable enough to start making updates.

I think a better approach would be to place all content, localized or not, in spreadsheets curated by the stakeholders. We could then find or build a simple tool to convert the spreadsheets' content into JSON. Only the JSON files would be deployed and a SPA framework like React or Angular could hydrate the DOM on the client side. However, that comes with its own difficulties. Google's static site

infrastructure makes it difficult to build SPA projects on top of the security reviews that make using third-party libraries a hassle to get approved. Also, How YouTube Works relies heavily on SEO, and client side SPA projects are more difficult to optimize for search engines.