The Stack That Helped Medium Drive 2.6 Millennia of Reading Time

Background

Medium is a network. It’s a place to share stories and ideas that matter — it’s where you move thinking forward, and people have spent 1.4 billion minutes — or 2.6 millennia — reading on Medium.

We get over 25 million unique readers every month and tens of thousands of posts published each week. But we want Medium to be a place where the measure of success isn’t views, but viewpoints. Where the quality of the idea matters, not the author’s qualifications. A place where conversation pushes ideas forward and words still matter.

I lead the engineering team. I was previously a Staff Software Engineer at Google, where I worked on Google+ and Gmail, and co-founded the Closure project. In past lives I’ve raced snowboards, jumped out of planes, and lived in the jungle.

The Team

I couldn’t be prouder of this team. It’s an awesome bunch of talented, curious, mindful individuals who come together to do great work.

We operate in cross-functional, mission-driven teams, so while some people specialize, everyone should feel able to touch any part of the stack. We believe that exposure to different disciplines makes you a stronger engineer. I wrote about our other values here.

The teams have a lot of flexibility in how they organize around their work, but as a company we set quarterly goals and encourage iterative sprints. We use GitHub for code reviews and bug tracking and Google Apps for email, docs, and spreadsheets. We’re heavy users of Slack — and slack bots — and many teams use Trello.

Initial Stack

We deployed to EC2 from the start. The main app servers were written in Node.js, and we migrated to DynamoDB for the public launch.

There was a node server that we used for image processing, delegating to GraphicsMagick for the actual hard work. And another server acted as a SQS queue processor for background tasks.

We used SES for email, S3 for static assets, CloudFront as CDN, and nginx as a reverse proxy. We used Datadog for monitoring and PagerDuty for alerting.

The site used TinyMCE as a foundation for the editor. Before launch we were already using the Closure Compiler and some portions of the Closure Library, but Handlebars for templates.

Current Stack

For a site as seemingly simple as Medium, it may be surprising how much complexity is behind the scenes. It’s just a blog, right? You could probably knock something out using Rails in a couple of days. :)

Anyway, enough snark. Let’s start at the bottom.

Production Environment

We are on Amazon's Virtual Private Cloud. We use Ansible for system management, which allows us to keep our configuration under source control and easily roll out changes in a controlled way.

We have a service-oriented architecture, running about a dozen production services (depending on how you count them and some more micro than others). The primary choice as to whether to deploy a separate service is the specificity of the work it performs, how likely dependent changes are to be made across service boundaries, and the resource utilization characteristics.

Our main app servers are still written in Node, which allows us to share code between server and client, something we use quite heavily with the editor and post transformations. Node has worked pretty well for us, but performance problems have emerged where we block the event loop. To alleviate this, we run multiple instances per machine and route expensive endpoints to specific instances, thus isolating them. We've also hooked into the V8 runtime to get insights into what ticks are taking a long time; generally it's due to object reification during JSON deserialization.

We have several auxiliary services written in Go. We've found Go very easy to build, package, and deploy. We like the type-safety without the verbosity and JVM tuning of Java. Personally, I'm a fan of using opinionated languages in a team environment; it improves consistency, reduces ambiguity, and ultimately gives you less rope to hang yourself.

We now serve static assets using CloudFlare, though we send 5% of traffic to Fastly and 5% to CloudFront to keep their caches warm should we need to cut over in an emergency. Recently we turned up CloudFlare for application traffic as well — primarily for DDOS protection but we've been happy with the performance gains.

We use a combination of Nginx and HAProxy as reverse proxies and load balancers, to satisfy the Venn Diagram of features we need.

We still use Datadog for monitoring and PagerDuty for alerts, but we now heavily use ELK (Elasticsearch, Logstash, Kibana) for debugging production issues.

Databases

DynamoDB is still our primary datastore, but it hasn't been completely smooth sailing. One of the perennial issues we've hit is the hotkey issue during viral events or fanouts for million-follower users. We have a Redis cache cluster sitting in front of Dynamo, which mitigates these issues with reads. Optimizing for developer convenience and production stability have often seemed at odds, but we're working to close the gap.

We're starting to use Amazon Aurora for some newer data, which allows more flexible querying and filtering than Dynamo.

We use Neo4J to store relations between the entities that represent the Medium network, running a master with two replicas. People, posts, tags, and collections are nodes in the graphs. Edges are created on entity creation and when people perform actions such as follow, recommend, and highlight. We walk the graph to filter and recommend posts.

Data Platform

From early on we've been very data hungry, investing in our analytics infrastructure to help us make business and product decisions. More recently we're able to use the same data pipelines to feed back into production systems to power data-driven features such as Explore.

We use Amazon Redshift as our data warehouse, providing the scalable storage and processing system our other tools build on. We continuously import our core data set (e.g. users, posts) from Dynamo into Redshift, and event logs (e.g. post viewed, post scrolled) from S3 to Redshift.

Jobs are scheduled by Conduit, an internal tool that manages scheduling, data dependencies, and monitoring. We use an assertion-based scheduling model, where jobs will only be executed if their dependencies are satisfied (e.g. daily job that depends on an entire day of event logs). In production this has proved indispensable — data producers are decoupled from their consumers, simplifying configuration, and the system is very predictable and debuggable.

While SQL queries running in Redshift work well for us, we need to get data into and out of Redshift. We've increasingly turned to Apache Spark for ETL because of its flexibility and ability to scale with our growth. Over time Spark will likely become the tool of choice for our data pipelines.

We use Protocol Buffers for our schemas (and schema evolution rules) to keep all layers of the distributed system in sync, including mobile apps, web service, and data warehouse. Using custom options, we annotate our schemas with configuration details like table name and indexes, and validation constraints like max length for strings, or acceptable ranges for numbers.

People need to remain in sync too so mobile and web app developers can all log the same way, and Product Scientists can interpret fields, in the same way. We help our people work with data by treating the schemas as the spec, rigorously documenting messages and fields and publishing generated documentation from the *.proto* files.

Images

Our image server is now written in Go and uses a waterfall strategy for serving processed images. The servers use groupcache, which provides a memcache alternative while helping to reduce duplicated work across the fleet. The in-memory cache is backed by a persistent S3 cache; then images are processed on demand. This gives our designers the flexibility to change image presentation and optimize for different platforms without having to do large batch jobs to generate resized images.

While it's now mainly used for resizing and cropping, earlier versions of the site allowed for color washes, blurring, and other image effects. Processing animated gifs has been a huge pain for reasons that should be another post.

TextShots

The fun TextShots feature is powered by a small Go server that interfaces with PhantomJS as a renderer process.

I always imagined switching the rendering engine to use something like Pango, but in practice, the ability to lay out the image in HTML is way more flexible and convenient. And the frequency at which the feature is used means we can handle the throughput quite easily.

Custom Domains

We allow people to set up custom domains for their Medium publications. We wanted single sign-on and HTTPS everywhere, so it wasn't super trivial to get working. We have a set of dedicated HAProxy servers that manage certs and route traffic to the main fleet of application servers. There is some manual work required when setting up a domain, but we've automated large swathes of it through a custom integration with Namecheap. The cert provisioning and publication linking is handled by a dedicated service.

Web Frontend

On the web, we tend to want to stay close to the metal. We have our own Single Page Application framework that uses Closure as a standard library. We use Closure Templates for rendering on both the client and the server, and we use the Closure Compiler to minify the code and split it into modules. The editor is the most complex part of our web app, which Nick has written about.

iOS

Both our apps are native, making minimal use of web views.

On iOS, we use a mixture of homegrown frameworks and built-in components. In our network layer, we use NSURLSession for making requests and Mantle for parsing JSON into models. We have a caching layer built on top of NSKeyedArchiver. We have a generic way to render items in a list with a common styling, which allows us to quickly build new lists with different types of content. The post view is built with a UICollectionView with a custom layout. We use shared components to render the full post and the post preview.

Every commit is built and pushed to Medium employees, so that we can try out the app as quickly as possible. The cadence of our release to the appstore is beholden to the review cycle, but we try to keep pushing as fast as we can, even if there are only minimal updates.

For tests, we use XCTest and OCMock.

Android

On Android, we stay current with the very latest editions of the SDK and support libraries. We don't use any comprehensive frameworks, preferring instead to establish consistent patterns for repeated problems. We use guava for all the things missing from Java. But otherwise, we tend to use 3rd party tools that aim to solve more narrow problems. We define our API responses using protocol buffers and then generate the objects we use in the app.

We use mockito and robolectric. We write high-level tests that spin up activities and poke around — we create basic versions of these when we first add a screen or to prepare for refactoring. They grow as we reproduce bugs to shield against regression. We write low-level tests to exercise the particulars of a single class — we create these as we build out new features and they help us reason about how our classes interact.

Every commit is automatically pushed to the play store as an alpha build, which goes out to Medium staff right away. (This includes another flavor of the app, for our internal version of Medium — Hatch). Most Fridays we promote the latest alpha to our beta group and have them play with things over the weekend. Then, on Monday, we promote it from beta to production. Since the latest code is always ready for release, when we find a bad bug, we get the fix out to production immediately. When we're worried about a new feature, we let the beta group play with things a little longer; when we're excited, we release even more frequently.

A|B Testing & Feature Flags

All our clients use server-supplied feature flags, called variants, for A|B testing and guarding unfinished features.

Misc

There are a lot of other things on the fringe of the product that I haven't mentioned above: Algolia has allowed us to iterate quickly on search-related functionality, SendGrid for inbound and outbound email, Urban Airship for notifications, SQS for queue processing, Bloomd for bloom filters, PubSubHubbub and Superfeedr for RSS, etc. etc.

Build, Test, Deploy Workflow

We embrace continuous integration and delivery, pushing on green as fast as possible. Jenkins manages all those processes.

Historically we've used Make for our build system, but we're migrating to Pants for newer projects.

We have a combination of unit tests and HTTP level functional tests. All commits have to pass tests before they can be merged. We worked with the team at Box to use Cluster Runner to distribute the tests and make this fast. There's nice integration with GitHub.

We deploy to a staging environment as quickly as we can — currently about 15 minutes — and successful builds are then used as candidates for production. The main app servers normally deploy around five times a day, but sometimes as many as 10 times.

We do blue/green deploys. For production we send traffic to a canary instance, and the release process monitors error rates before proceeding with the deploy. Rollbacks are internal DNS flips.