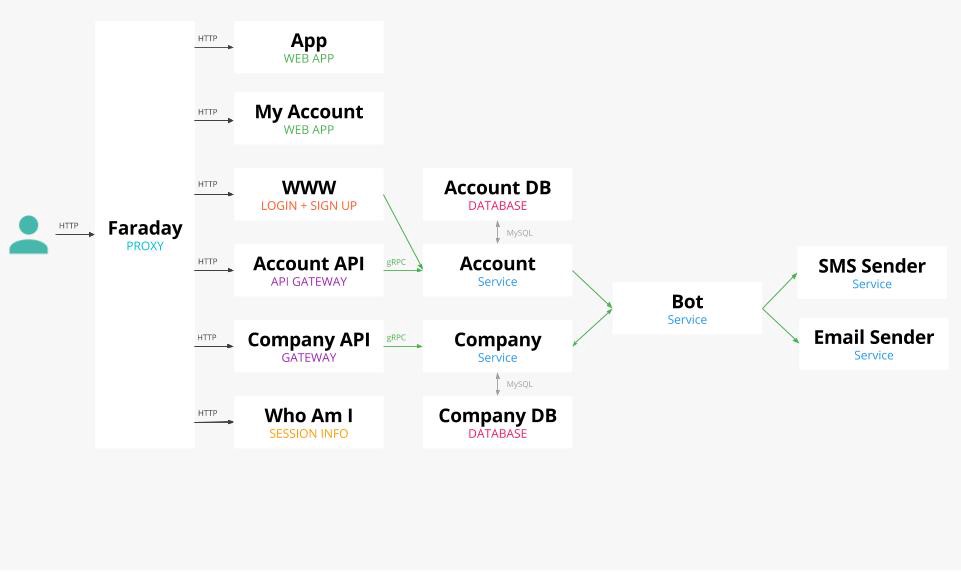
**Staffjoy’s V2 Architecture**



**Summary**

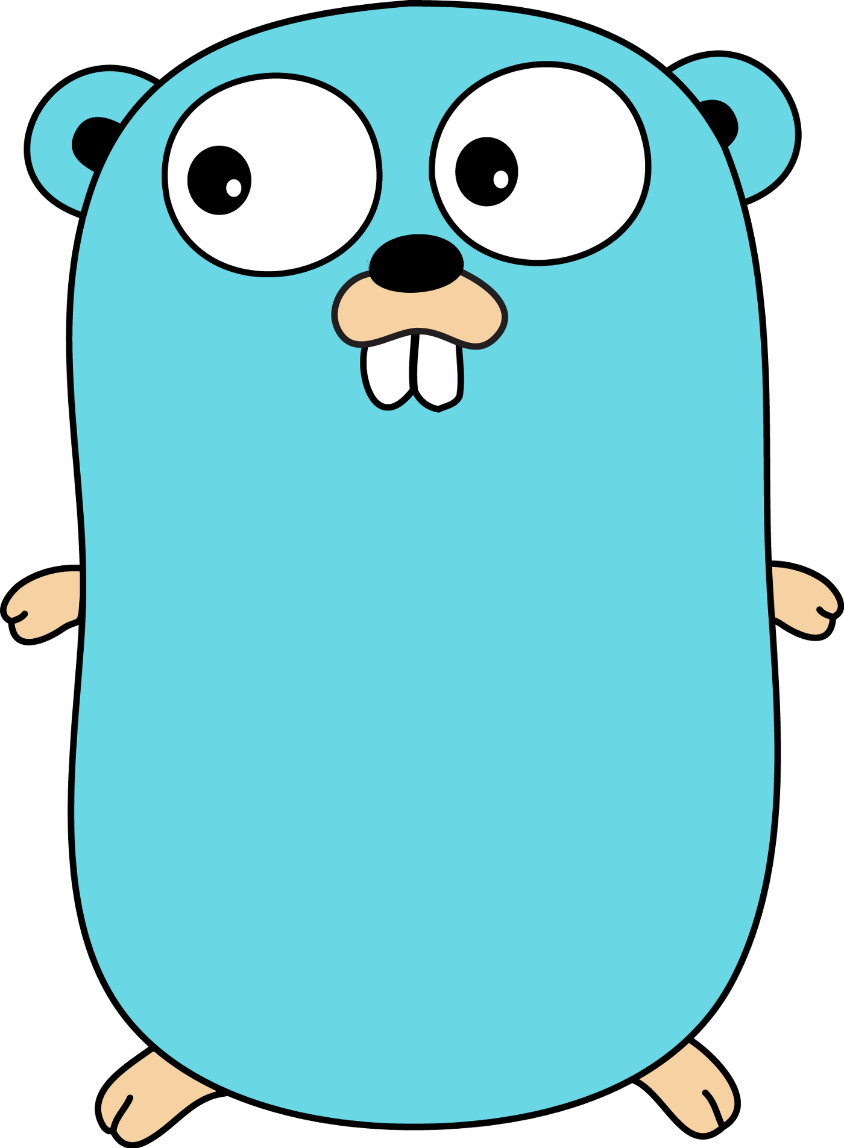
* Docker on Kubernetes using Google Container Engine
* Bazel build system
* Go microservice backend
* Redux front-end applications
* gRPC for internal communication
* Structured logging

**Background**

We built Staffjoy V1 using [Flask](http://flask.pocoo.org/) (covered more [here](https://blog.staffjoy.com/our-stack-dcb03fc8b4d8)). Things worked well — until we started to build more services and integrations. Each service required its own new repo, new CI jobs, and new servers. There had to be a better way!

Around July of this year, we decided to completely redo our product. We had to make some major changes to data normalization and deployments. We decided that starting fresh would be the best way.

With V2, we wanted to focus on messaging and integrations. We decided to use a microservice approach. One way we maintain simplicity is by using different databases for different services. That way, changes are less scary, and new services can use the best datastore for their needs.



Staffjoy’s backend services are all written in Go. (The Go gopher was designed by [Renee French](http://reneefrench.blogspot.com/).)

[Weak typing](https://en.wikipedia.org/wiki/Strong_and_weak_typing) caused most of our runtime errors in V1. To address this, we decided to use strong typing everywhere possible. So, V2 uses [protocol buffers](https://developers.google.com/protocol-buffers/), [Go,](http://golang.org) and [MySQL](http://mysql.com). (Our frontend uses Javascript, but I aspire to change it to [Typescript](https://www.typescriptlang.org/)!) Based on my Go experience at [OpenDNS](https://opendns.com), I thought that it would address our issues with Python. In particular, I like being able to execute asynchronous tasks without queues. This simplifies tasks such as sending email or executing callbacks.

To force ourselves to immediately think in microservices, we made a radical decision. We separated account logic and company logic into separate services. The services have separate MySQL databases, separated by a firewall. This forces us to rely on gRPC instead of SQL JOIN statements to get a worker’s name.

**Development and Build System**

To start, we chose to use a monorepo. We store all code for V2 in a single [Github](https://github.com) repo. Logic becomes centralized, which allows for easier refactoring across the code base. It also allows us to provision new services using committed configuration files.

Google recently open-sourced [Bazel](https://bazel.build), a build system inspired by their internal Blaze system. It makes monorepos fast by caching builds. Bazel also understand how code changes invalidate its caches. This means that building our 14 Docker images normally takes less than a minute. Bazel works smoothly with Go and its dependencies. But, we still have not figured out how to best handle JavaScript dependencies.

When you publish a shift on Staffjoy, we send a text message to each worker alerting them of their schedule. The process of changing a shift can involve as many as six different microservices. To aid development, we want our developers to be able to run the complete environment on their laptop. We chose to run run Kubernetes inside of [Vagrant](https://www.vagrantup.com/) for development purposes. One reason we did this is that we rely on Kubernetes’ DNS for service discovery. To save RAM, we sometimes JavaScript code against our staging environment.

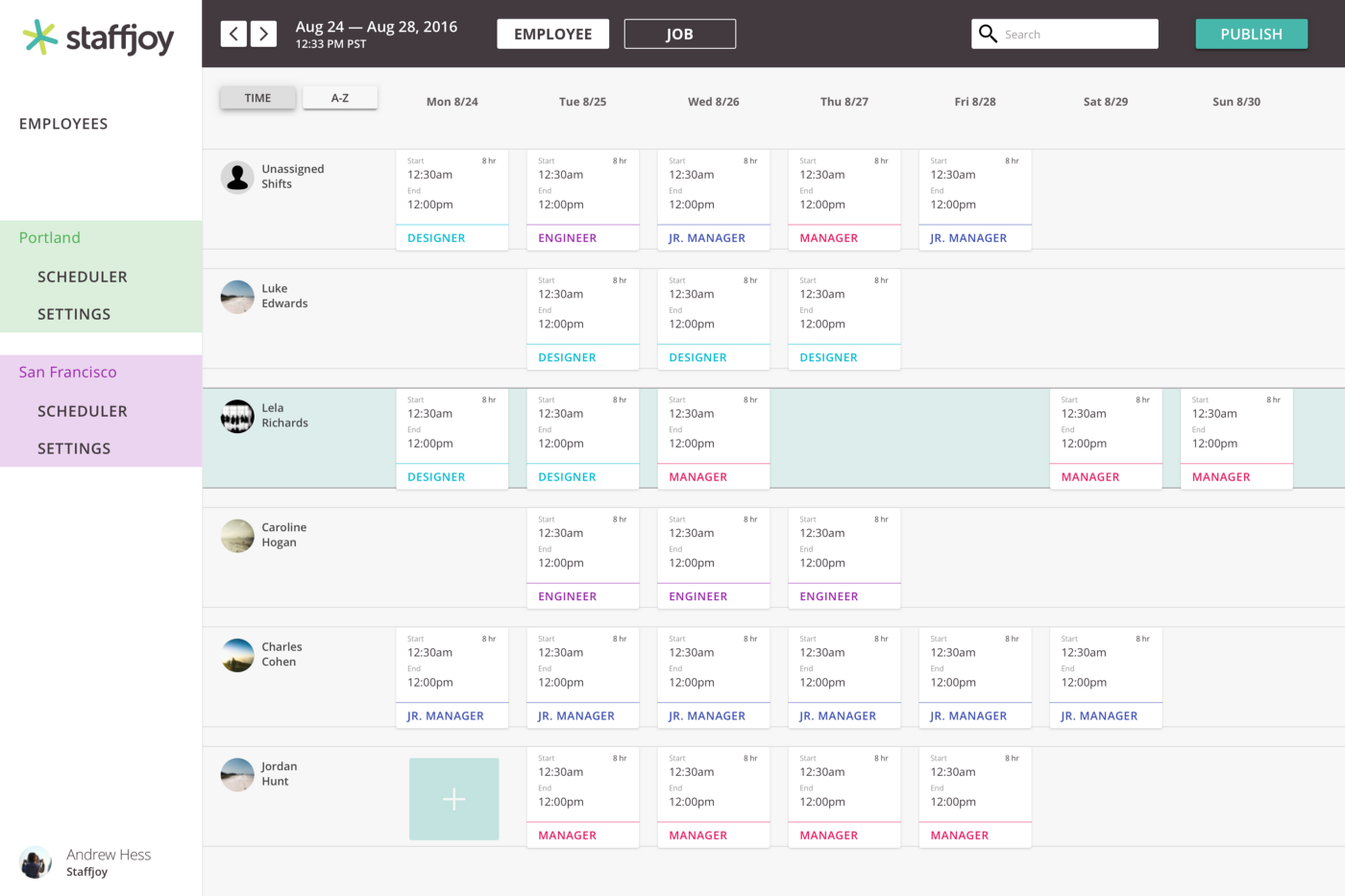
**Faraday — Authentication and Proxy**

When a user logs in, we set a JSON web token. We built a proxy layer called Faraday to manage user authentication. Faraday reads HTTP request, then sets internal metadata about the authentication and authorization. This allows internal services to identify users with minimal code. Faraday also routes requests to the correct service inside Kubernetes.

Faraday is the only Staffjoy service that faces the Internet. Because all web traffic goes through it, we added in some convenience features. For instance, Faraday improves security in a variety of ways. It redirects HTTP traffic to HTTPS, and it sets HSTS headers. Every request has its latency measured by Faraday and reported to [Google Cloud Trace](https://cloud.google.com/trace/). In the future, we plan to continue improving Faraday. For example, we plan to standardize favicons through it.

**Web Apps in Redux**

Our front-end applications use [Webpack](https://webpack.github.io/), [React](https://facebook.github.io/react/), and [Redux](https://github.com/reactjs/react-redux). These libraries have made it straightforward to build complex interfaces, such as draggable cards. In development, we use [react-storybook](https://github.com/storybooks/react-storybook) to build components.

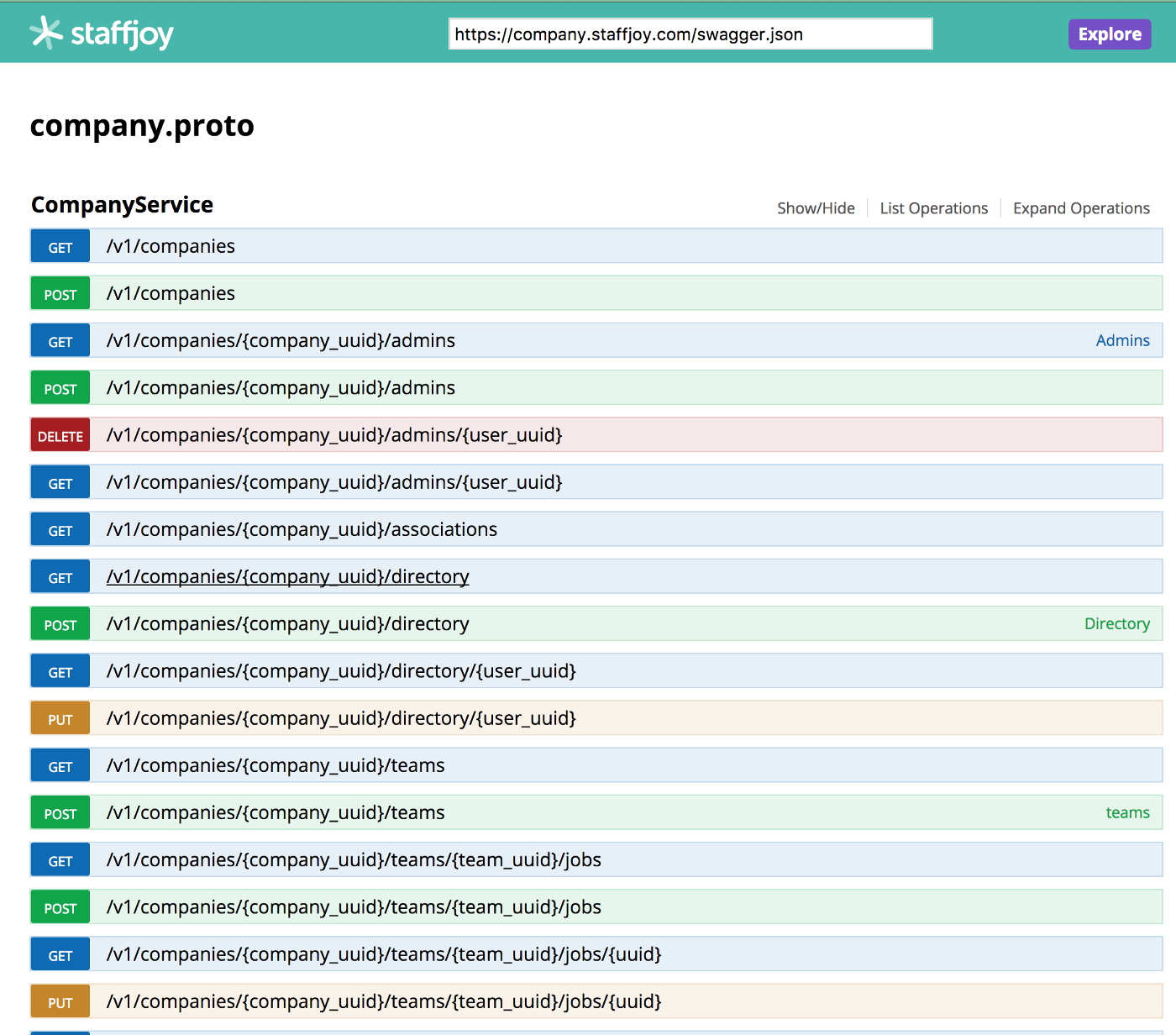


Staffjoy Scheduling UI

**Protocol Buffers and gRPC**

If you’re used to a JSON over HTTP REST API, then Protocol Buffers with gRPC serve a similar purpose. Protocol Buffers are a strong-typed serialization tool from Google. The library generates clients in a variety of programming languages. gRPC is a messaging protocol over HTTP2. Instead of REST endpoints, gRPC defines functions on Protocol Buffer message types.

To build a REST API for use by our front-end applications, we use [grpc-gateway](https://github.com/grpc-ecosystem/grpc-gateway). It lets us annotate gRPC functions to map them to HTTP routes. It even auto-documents every endpoint using [Swagger](http://swagger.io/)- check out our [company docs](http://company.staffjoy.com/ui/)).



*grpc-gateway automatically generates a Swagger specification for the API*

We use the [Gorp](https://github.com/go-gorp/gorp) library where possible to map Go structs to database columns for simple CRUD. To use Protocol Buffers with Gorp, we need to add tags to the generated protocol buffer structs. We do this using [GoGoProto](https://github.com/gogo/protobuf), a library that extends Protocol Buffers.

Finally, we use [Logrus](https://github.com/sirupsen/logrus) for structured logging. Stackdrivers’ logging interface makes it easy to query structured logs. We have already implemented a full audit log for keeping track of all changes to the database.

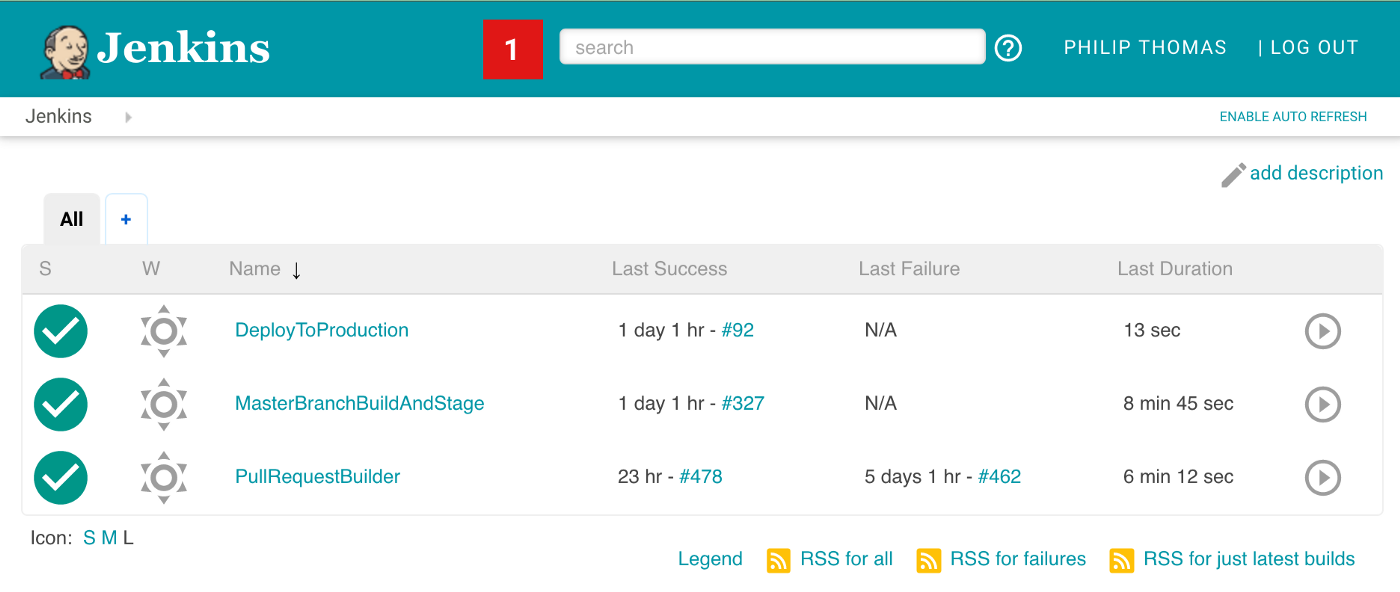
**Building a Bot**

To tell workers when they have shifts, Staffjoy communicates through text messages. In our user studies, we found that many restaurant managers share schedules over SMS. We aim to replace this manual workflow with an notification bot.

We started with a simple bot, and we improve it almost every day. We chose to make the messaging logic agnostic of the communication layer. This means that we can add new messaging systems in the future.

Using Go, we send asynchronous messages about shift changes to the bot service. The bot service then decides how to notify workers. In the future, we plan to build in better state management. We also want workers to be able to query the bot.

**Continuous Integration**



Clean Jenkins with only three jobs

We use Jenkins to test, build, and deploy Staffjoy’s Git repo. We only have three jobs:

* Pull request tester
* Build, test, and deploy changes from the “master” branch to our staging environment
* Promote builds from staging to production

**Conclusion**

Microservices took time to configure, but have already been helpful. Most backend services now have three or fewer Go files. This simplicity reduces the overhead of making changes or adding new services.

If you’re interested in hacking on microservices, messaging, and scheduling — [we’re hiring](http://staffjoy.com/careers/)!