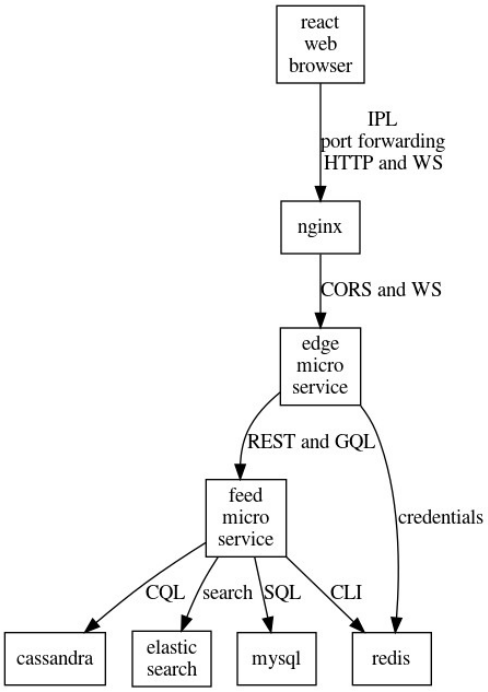
Ionic vs React vs Flutter

by Glenn Engstrand

It is relatively easy for any organization to standardize on a single back end tech stack. That can be a good thing as it reduces fragmentation issues with employing and managing engineers with different skill sets. Front end development is very different in that regard. There are many different form factors that you will need to support in order to deliver a rewarding experience to a diverse audience. Learning any tech stack takes both time and mind. Staffing up on a tech stack is a big commitment to any organization. Which front end tech stacks should you commit to in order to achieve the best long term benefits?

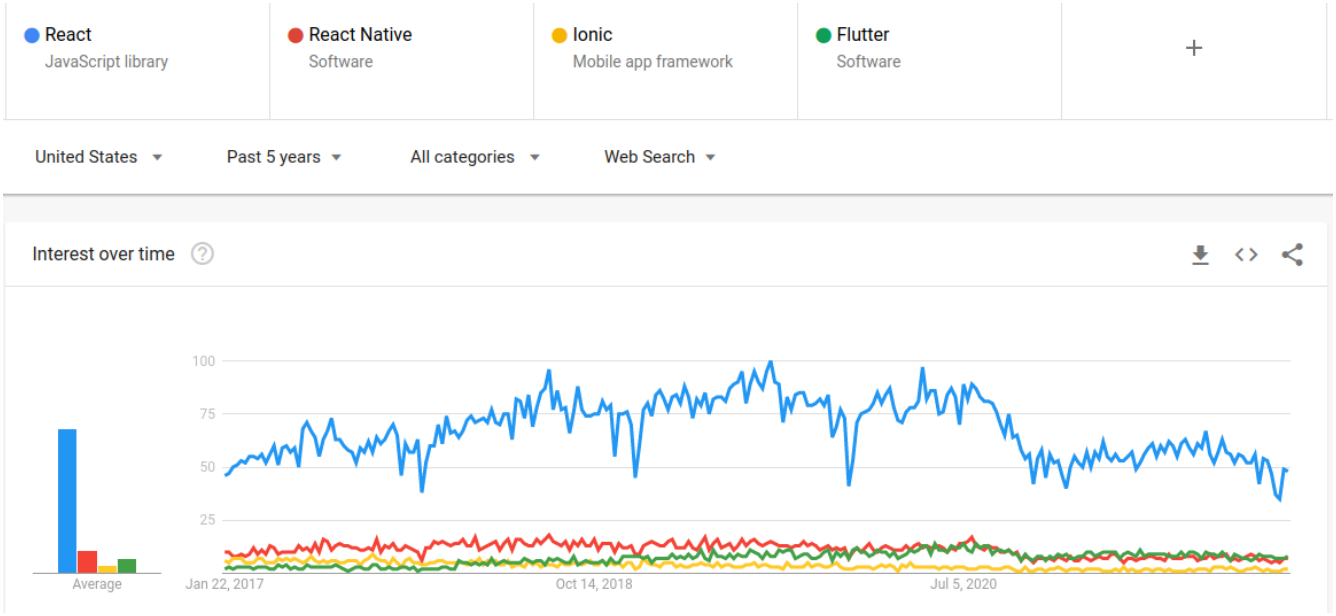
If you have visited this blog before, then you already know that I implement feature identical rudimentary polyglot persistent news feed microservices in various programming languages and tech stacks then compare them with previous implementations in terms of architecture, design, coding, and performance under load. To that end, I currently have thirteen implementations of that same back end service.



But what about the front end? I also evaluate front end tech stacks by implementing a GUI for this already coded back end news feed service. So far, I have three implementations that I will cover in this blog that you are reading now. I will also cover some test environments that I use for the news feed. Afterwards, I will share what I discovered and lessons learned.

Who's Calling?

What are the different programs that I have coded to call the news feed service?

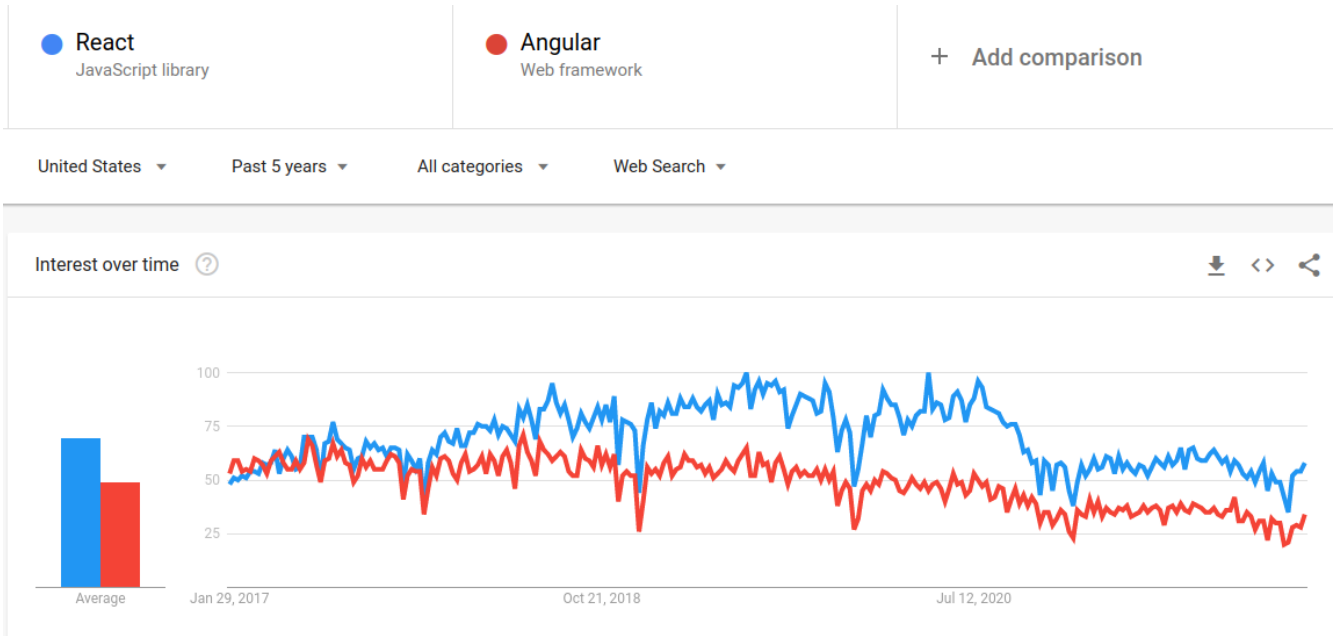


Of course, each service implementation comes with its own unit tests. I have a Gatling test that is suitable for local profiling. I also have a separate app, written in Clojure, that I use for test automation of the service in Kubernetes. Depending on how you run it, the program either performs load testing or integration testing. Normally, an integration test just makes sure that each end point satisfies its contractual agreements. Since these implementations all share the same underlying data stores, this integration also tests to make sure that the data at rest is what

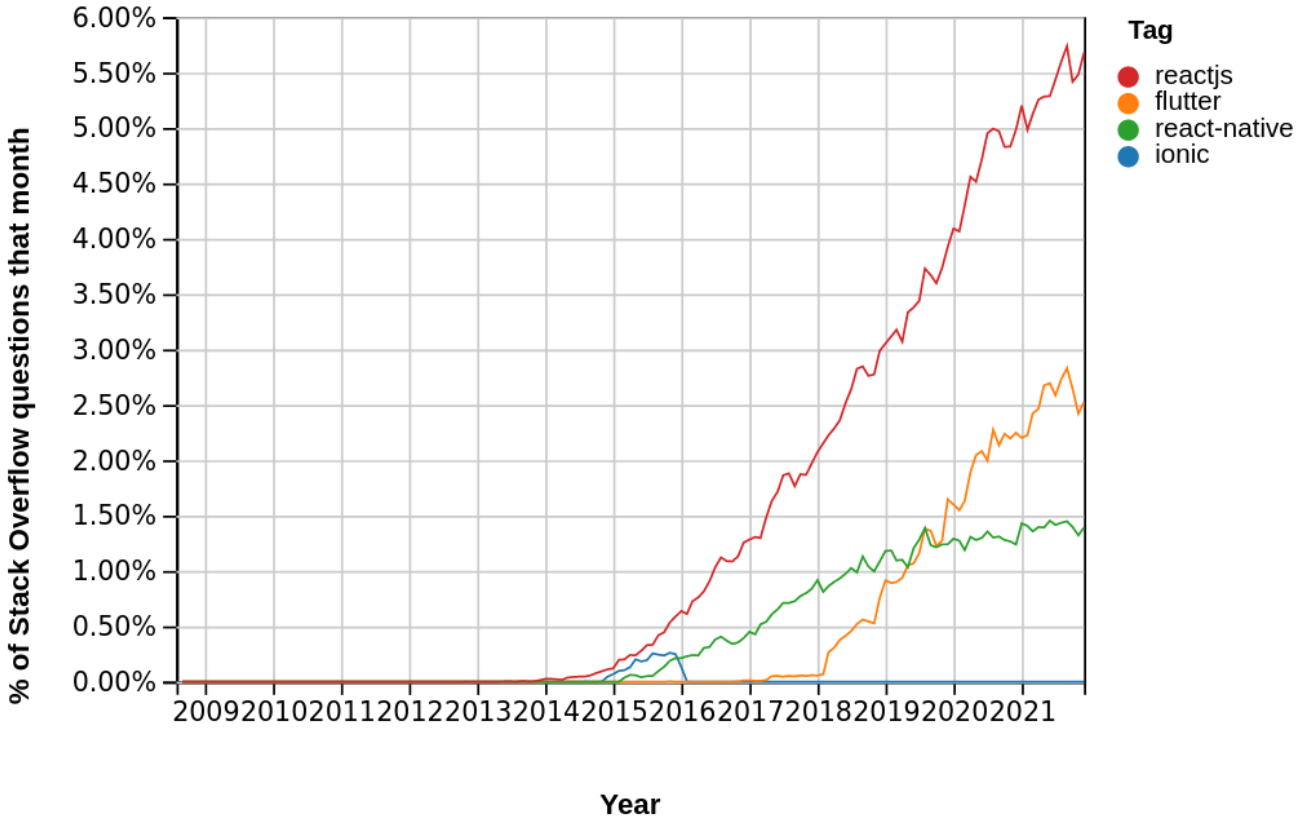
was expected. When I am deving, I typically run the integration test from my laptop while port forwarding to the service in Kubernetes. The load test spins up 3 threads forever creating 10 users, friending each user on average 3 times then posting 10 outbound stories (of 150 words each) for each user. I usually let that run inside the cluster for a couple of hours, then collect the performance results for further comparative analysis.

I implemented a web client using React which is a free and open-source front end JavaScript library for building user interfaces based on UI components. It is maintained by Facebook and a community of individual developers and companies. React can be used as a base in the development of single-page or mobile applications.

This implementation is a single-page app (SPA) written in TypeScript which gets transpiled to JavaScript and eventually downloaded then run on the web browser. It depends on the following popular libraries; Redux, Router, and Axios. Redux is a predictable state container for JavaScript apps. Router provides declarative routing for React apps at any scale.



Axios is a promise based HTTP client for the browser and Node. There is no coverage of nextjs here so the SPA runs exclusively in the web browser but you can choose to host its files either locally on your laptop or inside your Kubernetes cluster.

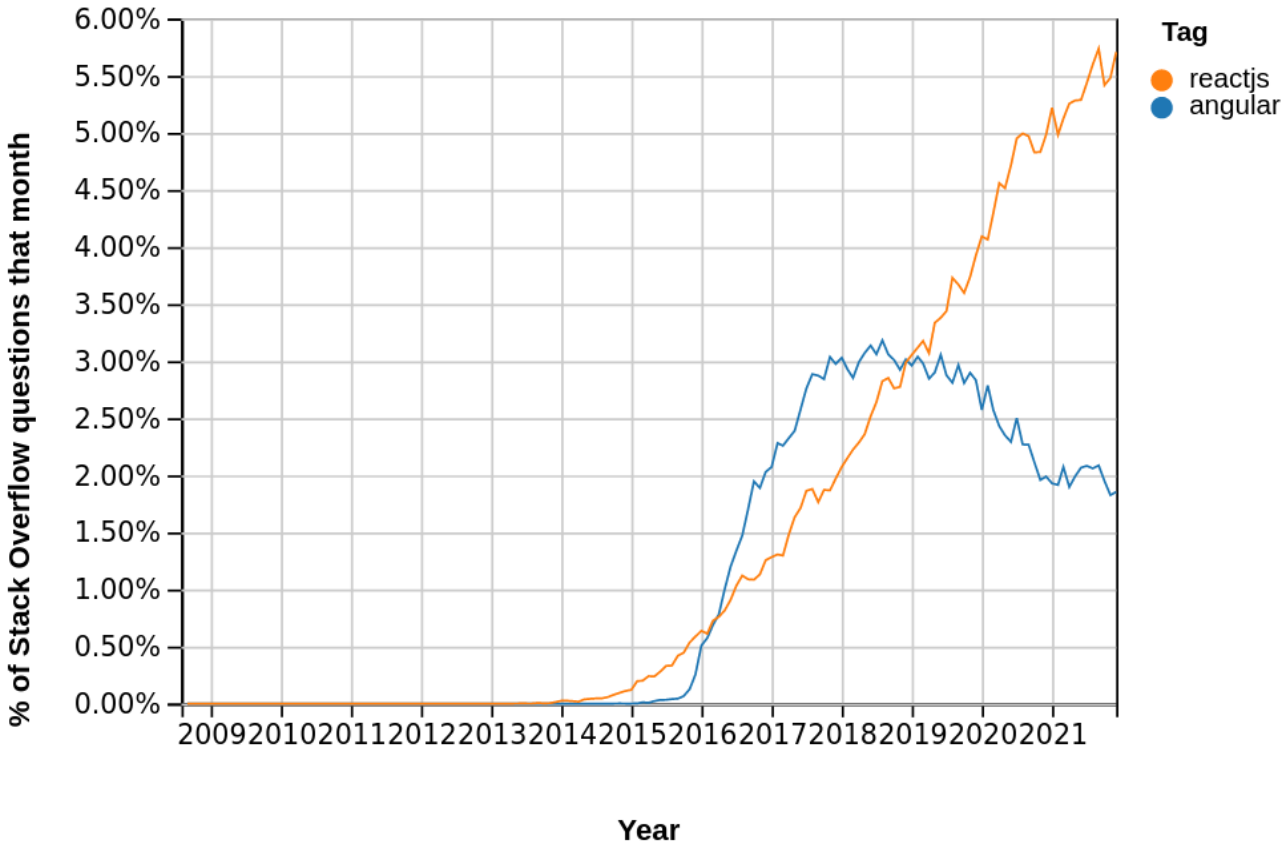


I have also written a couple of cross platform mobile front ends. There are some advantages to cross platform development. You need only to staff up on a single front end skill set. Most organizations already have web developers so that would be the most preferred skill set to develop cross platform mobile on. You would have but a single code base to develop on in order to publish apps for both Android and iOS mobile devices. There are also some disadvantages too. The user experience on cross platform mobile apps is more clunky than native apps. Cross

platform apps are not as good as native apps when it comes to specific hardware integration.

The first client was built near the end of 2017 on top of the Ionic framework. Ionic is an open source UI toolkit for building performant, high-quality mobile and desktop apps using web technologies (HTML, CSS, and JavaScript) with integrations for popular frameworks. The programming language is TypeScript.

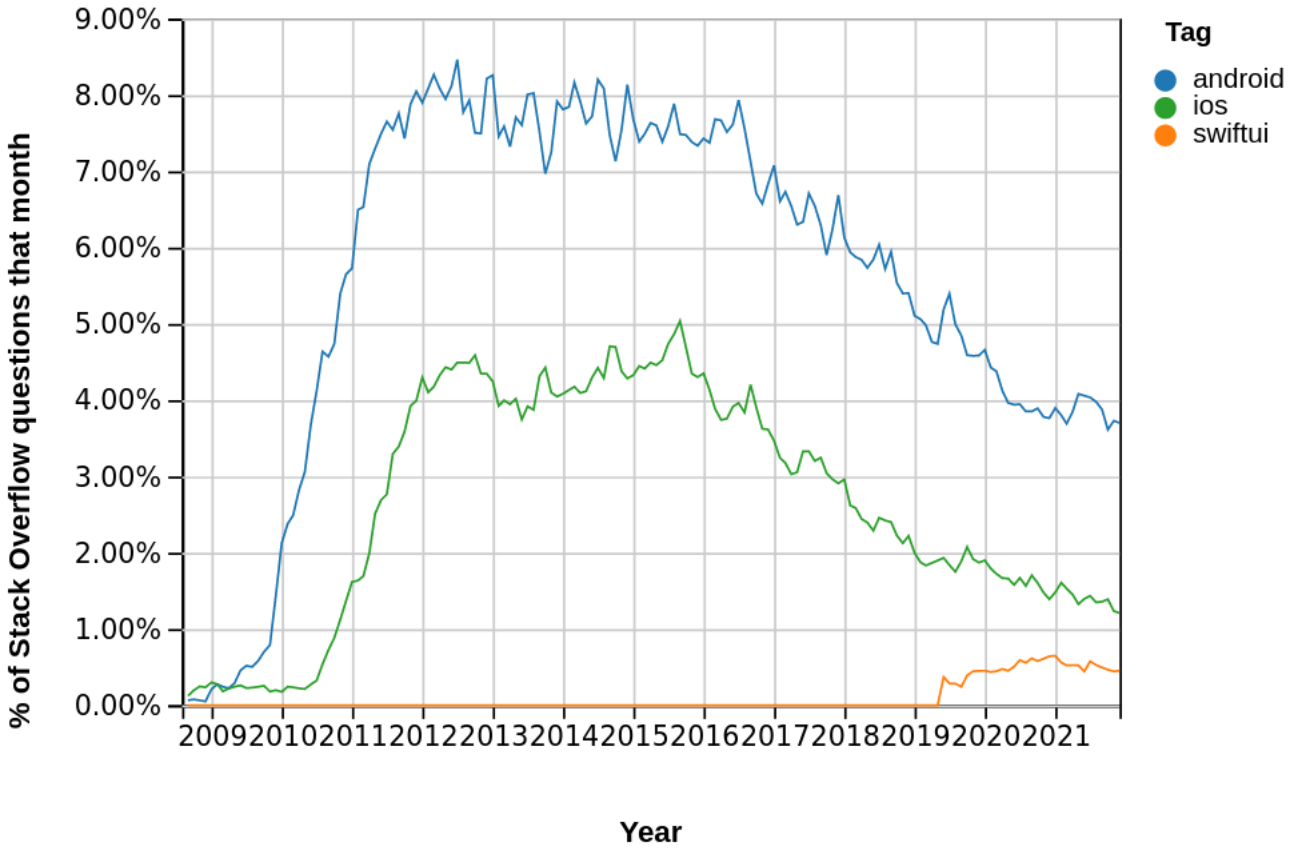
I used Angular at the time because that was what was available and the mobile application development framework that Ionic integrated with was Apache Cordova. Today, you can develop Ionic apps in Angular, React, or Vue and Cordova has been replaced with Capacitor so some of my coverage here may not accurately reflect on your experience were you to create a new Ionic project in 2022 or later.



The second client was built recently using Flutter which is an open source framework by Google. Flutter transforms the app development process. Build, test, and deploy beautiful mobile, web, desktop, and embedded apps from a single codebase. The programming language is Dart.

Commonality

The React app and the Flutter app have some things in common. They both use the Material UI design which is an open source adaptable system of guidelines, components, and tools. Material UI is a good choice for a professional yet generic look and feel GUI without a lot of effort. These two apps both integrate with the edge service which provides oauth2 and a GraphQL facade over the news feed service for querying.



With regards to authentication, the React app uses the implicit grant flow while the Flutter app uses the password grant flow. More on that later.

All three front end apps are internally designed on some variation of the MVC or Model View Controller pattern. View components are responsible for generating an object hierarchy of UI delegates that can be linked to either the DOM (Document Object Model) for web or OS specific native mobile widgets. Angular creates the DOM directly but React creates a virtual DOM which gets synchronized with the DOM.

Flutter UI delegates draw directly to the canvas. This approach makes for a quicker repaint time but be advised that single code base cross platform Flutter apps don't look native at all. For Ionic and React, the UI delegate hierarchy gets built declaratively using markup. Although the official documentation claims otherwise, procedural code is executed in order to deliver the UI delegates for Flutter.

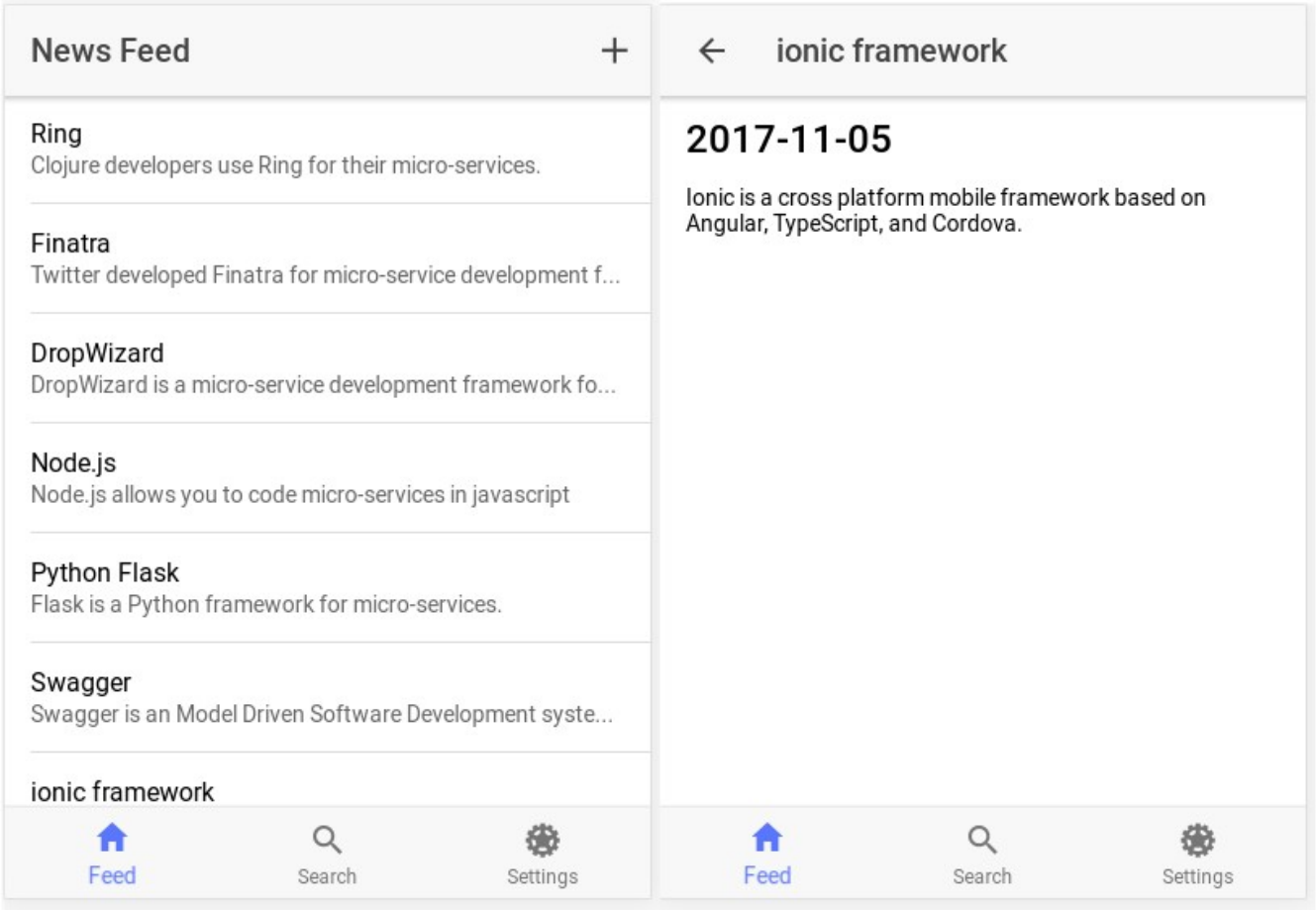
Discovery

There was plenty of discovery encountered while developing these front ends.

Ionic is a framework that is a combination of various technologies that were not originally intended to work together. There is a lot of complexity due mostly, but not entirely, on Angular and SCSS. It is not always readily apparent how everything is interconnected.

Of the three front end implementations for the news feed service, the Ionic implementation is incomplete and yet has twice as much code as the others. This code bloat is partially due to the accidental complexity of the franken framework itself but also partially due to the overzealous starter template that I used.

In terms of developer popularity, Ionic originally showed a lot of promise. Time has revealed that promise to be somewhat unfulfilled. It has achieved nowhere near the level of adoption that I had originally expected. This project doesn't even build anymore due to all the changes to that framework over time and that a backwards breaking change was introduced to a minor version upgrade of SCSS (which is part of the Ionic stack) that makes it incompatible with recent versions of Node (also a part of the Ionic stack).



The edge service is specifically designed to accommodate front end apps. It only stands to reason that there would be tight coupling between each front end app and the edge service.

I picked golang for the edge service so that I could quickly cobble together three different OSS projects in order to provide the necessary functionality but, as of the time of this writing, these repos are immature and have bugs. The graphql-go project is not as readily composable as Apollo. The code response type is not working properly in the go-oauth2 repo.

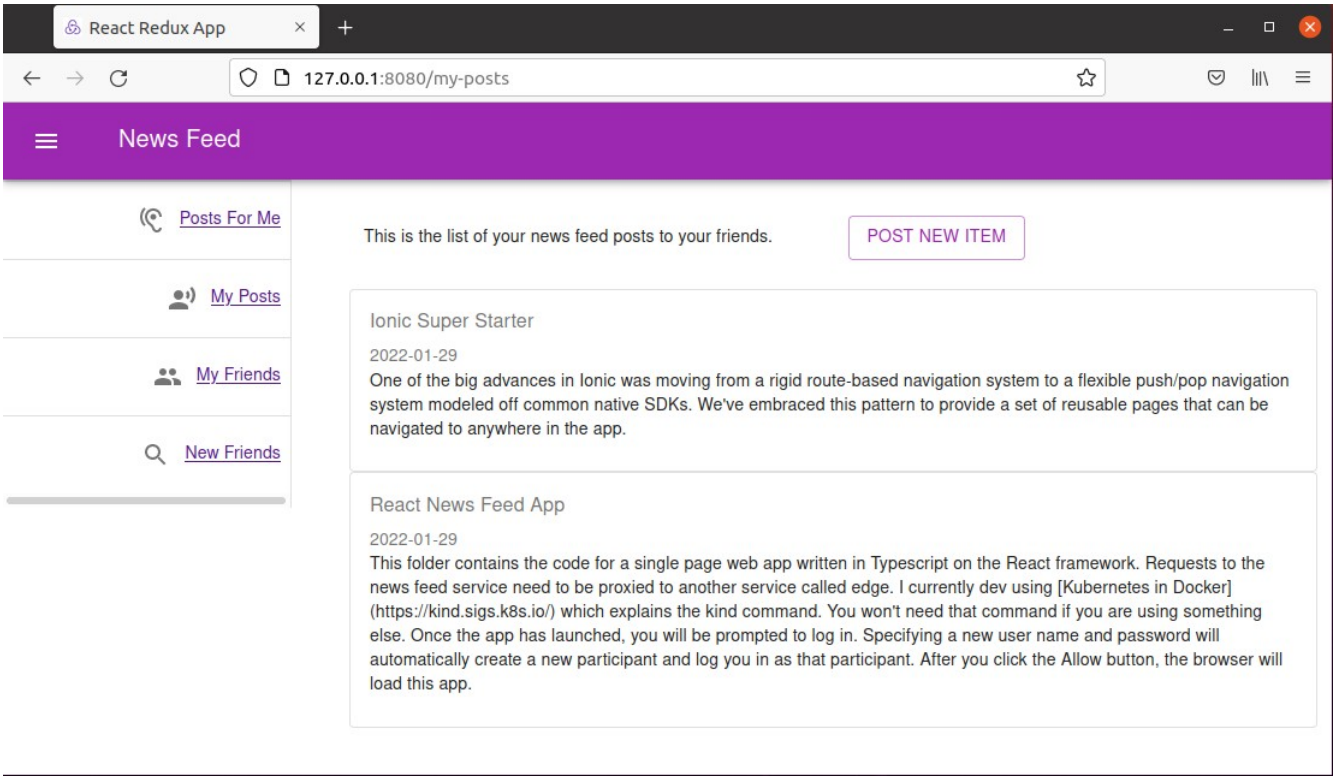
This version of go-oauth2 depends on a couple of packages that are now considered as having security vulnerabilities. Upgrading to a newer version is not trivial. The gorilla websocket repo does not fully comply with the websocket standards around the connection and upgrade headers and does not permit the use of the authorization header.

Front end web developers expect to dev with a very streamlined code and test cycle. They want to be able to dev locally so that they can simply refresh the web browser in order to pick up recent changes. For mobile developers, the Android emulator and the iOS simulator can be perceived as slow which can be aggravating.

The rule of hooks in the React web framework takes some getting used to and the payloadCreator callback from the createAsyncThunk in the react redux toolkit does not work exactly as documented when it comes to passing in arguments.

Of the three front end stacks, Flutter is the newest technology. It should come as no surprise that it is also the least mature. Eventually, the APIs will settle down but the changes can be hard to adjust for in the meantime. This API churn also means that a lot of the documentation online is no longer up-to-date which means it won't work if you simply copy then paste the sample code into your project. Of the three implementations, the Flutter version required the least amount of code.

From an architectural perspective, layering up a GUI is a synchronous operation and yet calling a back end service (used to populate data for the GUI) should be an asynchronous operation. This impedance mismatch needs to be accommodated for in every front end tech stack. The least complicated way to handle that in Flutter is to use the FutureBuilder. That class should be able to work with any type of Widget but at the time of this writing it works only with a Scaffold widget. The scaffold is the top level or root widget for a screen which provides additional capabilities such as an app bar, a drawer, or floating action buttons.



Flutter claims to be the one framework that you can use for all front end development including iOS, Android, Mac OS X, Linux, Windows, and web. At the time of this writing, webviews are broken when running on the chrome device (i.e. web). A mobile app can use a webview in order to use the same oauth2 grant flow as a web app. This can be more secure as the mobile app

code never sees the user's password. Because the webview does not always work, the mobile app has to collect the user's credentials and use the less secure oauth2 password grant flow instead.

As mentioned earlier, all three of these front end apps are based on the MVC pattern. A significant amount of controller code is dedicated to binding the view to the model. It is prudent to limit that amount of code by using a common library to that purpose. For React, I used the Redux library. The other popular approach for data binding in React is Context which comes bundled with it. Redux is more complex but does a better job of handling a large velocity of data changes with a smoother repaint.

The inbound fragment shows messages from other participants (via web socket) which could be coming at a large velocity. For Flutter, I used the Provider library. There is a language feature in Dart, known as null safety, where compile time checking prevents null pointer exceptions at runtime. The Provider library doesn't support null safety so I had to disable that language feature.

The unit tests for the React SPA do not attempt to mock the Redux store. The service calls fail silently and you end up with the default store state of a new user. The actual binding logic is not covered. The official recommendation is to mock the Axios calls which I did not do because I ran into build errors when I tried to add Jest.

The unit tests for the Flutter app do mock the providers which get wired up in a test harness version of the top level app. The binding logic gets covered but not the actual top level app. I gave up on the Ionic version before I got to unit tests so there are no unit tests for that app.

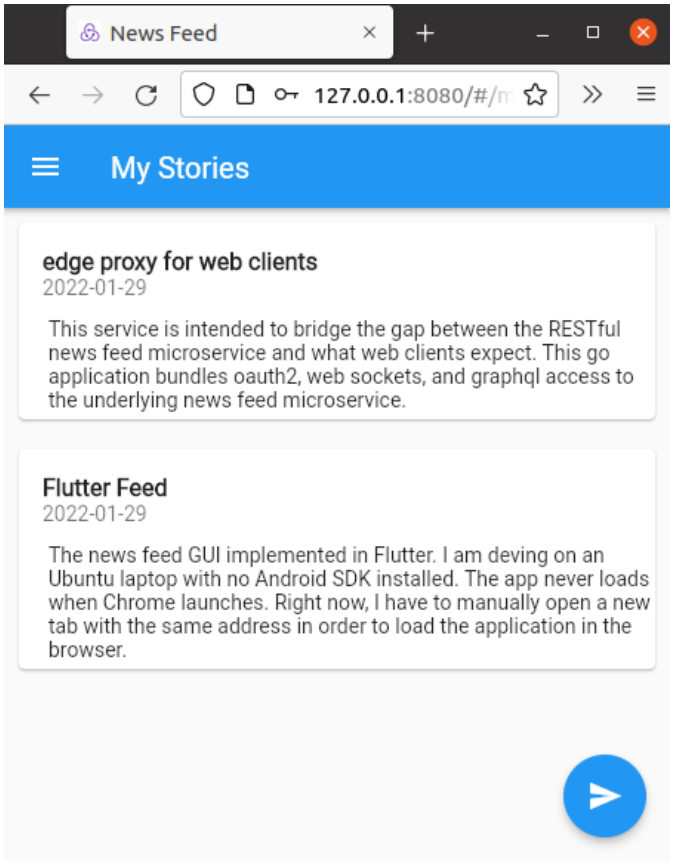
Testing this system with the thirteen different implementations shows just how easy it is for the integration test to get out-of-date in terms of test coverage, especially when it comes to the occurred and link properties.

Lessons Learned

Here are my best practice recommendations that came out of developing these front ends.

For the scope of this work, I bundled the handling of all of the edge concerns into a single service but that is less than optimal. The GraphQL parts are tightly coupled with the specific type of front end app but the oauth2 and single sign-on parts should be the same for all types of clients. The latter could be more general purpose in nature.	It should be designed as a platform service and developed using either off-the-shelf technology and / or your back end developer friendly tech stack. Since the former is tightly coupled with the front end, consider using front end developers to maintain it so it should be built with a front end developer friendly tech stack such as TypeScript on Apollo hosted by Node. The Apollo project is a very mature and stable framework for GraphQL.

For web, use Nginx when deving in Kubernetes or when running in higher environments such as production or UAT. This approach allows for a separation of CORS concerns from each back end service or even the edge service. Use react proxy (hosted by Node) when deving locally in React. The Flutter command line gives you the ability to dev locally either as a web app or via the Android emulator. Flutter also gives you the ability to build and package your app for the web, for desktop (beta), and for both app stores.



In terms of push notification capability, use either an external or embedded message bus or a distributed log on the back end for a more scalable solution. On the front end, consider leveraging GraphQL subscriptions which are based on the web socket protocol.

I ran into build errors when I tried to add Jest.

Get in the habit of updating your test automation (unit tests and / or integration tests) every time you change the service implementation. A PR with changes to one but not the other should automatically be considered suspect.

Crystal Ball Architecture

Wouldn't it be nice if you could predict the future when picking a tech stack to invest in? I would never have evaluated the Ionic framework if I knew what it's future was to become.

Will Flutter suffer the same fate? It is backed by a company with deeper pockets but also a company known to abandon projects that don't get developer mindshare quickly enough.

Cost conscious companies choose cross platform mobile development because they can just assign the work to web

developers but Dart is not TypeScript and Flutter is neither React nor Angular. Even the most experienced web developer will need time to become proficient in Flutter and Dart.

On the other hand Flutter is well thought out, easy to use, has convenient packaging options, and can be used to build professional, elaborate GUIs with less code. Will web developers want to learn Flutter and Dart? Only time will tell.