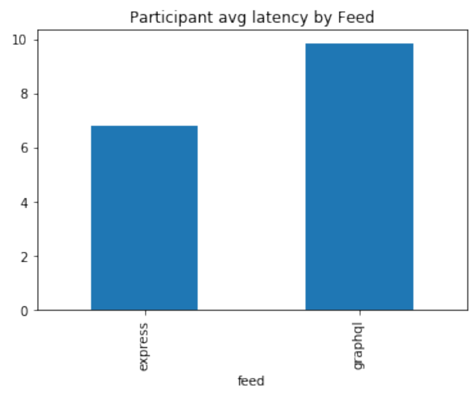
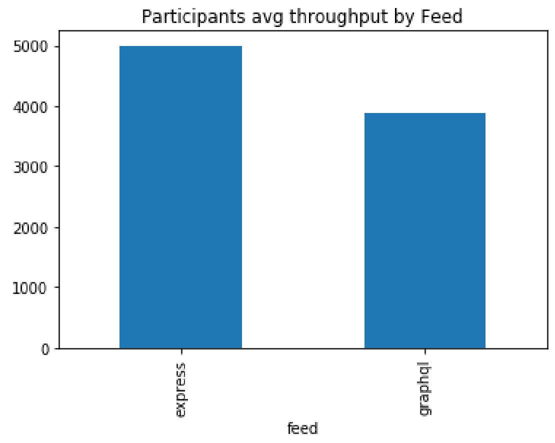
GraphQL vs REST

by Glenn Engstrand

If you are in a technology company whose application stack includes Node.js and you are considering writing your next microservice in GraphQL and / or TypeScript, then this article is for you. I developed, evaluated and compared two microservices; one in GraphQL and TypeScript, the other in Express and JavaScript. Here we cover that analysis in terms of architecture, historical context, design, code, and performance under load.



The predominant approach to modern API design is known as REST which was introduced by Roy Fielding in 2000. In 2015, Facebook introduced a competing approach called GraphQL which has steadily risen in popularity since then. There are about two technology conferences per year devoted to GraphQL. Depending on how you measure this, GraphQL currently gets about a quarter to a half as much attention as REST. For those of you who are into "Crossing the Chasm" style technology adoption trends, GraphQL is now in what is known as the Early Adopter phase. With REST, you typically code handlers for each path and method combination. Each handler creates the entire response to each request. With GraphQL, you define a single schema. You are supposed to code resolvers for each field then let the framework stitch the individual field responses together in order to form a coherent response to the inbound request.



Microsoft first released the TypeScript programming language in 2012. TypeScript is transpiled to JavaScript which gets run. What makes TypeScript more interesting than JavaScript is that static type checking transpiler which lets you catch more bugs in your code sooner. TypeScript was originally intended to be run in the web browser. I first developed the Node based news feed microservice in January 2017 but I chose JavaScript back then because TypeScript wasn't really ready yet for running server-side in Node. Interest in TypeScript is still a small fraction (one tenth to one fifth) when compared to JavaScript. In terms of technology adoption, it is considered to be in the Early Majority phase most probably due to it being the default language for the Angular web framework.

I thought that now might be a good time to see how mature server side TypeScript is in terms of microservice development. I also wanted to explore GraphQL in order to fully understand its merits. So I developed a rudimentary news feed microservice that is similar to my original JavaScript microservice based on the Express framework (feed 4) only this time in TypeScript and based on the Apollo Server for GraphQL (feed 10). All of this code is open source that you can find in my personal github repo. I needed to evaluate feed 10 under load so I also had to enhance the load test application (load) and the API gateway (proxy) in my test lab because GraphQL APIs are very different from RESTful APIs.

Architecture

In order to better understand the relative merits of these two microservices, let us compare and contrast their respective software architectures.

We want to be able to make a valid comparison between feed 4 (Express and JavaScript) and feed 10 (GraphQL and TypeScript) so both microservices need to be as feature identical as possible. This means that there are a lot of similarities between the two implementations.

Both microservices are polyglot persistent. They both use MySql fronted by Redis for participants and friends, Cassandra for inbound and outbound feed items, and Elasticsearch for keyword based search. Both microservices run in Node which has embedded within it Google's open source V8 engine for running single-threaded JavaScript.

There are also profound differences in the architectures of these two microservices.

Node provides the application with a single-threaded runtime so every method call has to be non-blocking. For feed 4, I used the callback mechanism where each function is passed as part of its arguments two functions for handling success and error results. For feed 10, I used the async and await mechanism which depends on promises.

Having to rewrite a low-level component in order to make something as fundamental as unit tests working is not a good sign in terms of software maturity.

The API design for feed 4 is somewhat RESTful. In the URL for each API call, the path identifies the main entity; participants, friends, inbound, and outbound. The HTTP method is POST for creates and GET for fetches.

Here is an illustrative REST example which fetches the inbound feed items for participant 4 from feed 4.

```
curl ${FEED_URL}/inbound/4
[{“occurred”:”2019-11-02”, "subject":"test subject", “story”:”test story”}]
```

The GraphQL API design for feed 10 consists of three mutations and two queries. There is a single schema where friends, inbound, and outbound are all attributes of each participant.

In this example request, the query as before is shown only this time in GraphQL. Notice that the request specifies what should get returned in the response.

```
curl \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{ "query": "query { participant(id: 4) { inbound { occurred, subject, story } } }"}' \
  $FEED_URL
{"data":{"participant":{"inbound":[{“occurred”:”2019-11-02”, "subject":"test subject",
“story”:”test story”}]}}}
```

Historical Context

In the days before REST, API design was RPC style where every endpoint was basically a Remote Procedure Call. Like America’s wild, wild West period in history, this was a very chaotic time for APIs. Developers designed them in order to expedite the immediate feature at hand. Typically, this included introducing hidden side effects to the APIs such that you had no idea if their behavior was a feature or a bug. What REST did was introduce some basic rules of the road that made it easier to reason about APIs, to learn them, and to write test automation for them.

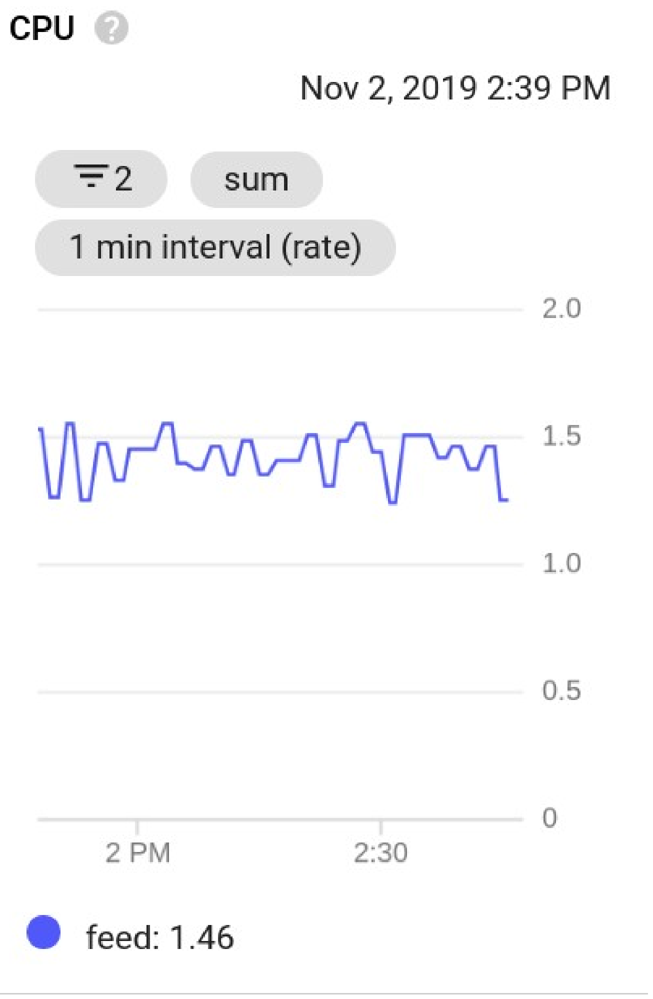
For a large enterprise with a complex business model, RESTful APIs tend to have a large number of endpoints. Without adequate and up-to-date documentation, it could be difficult to figure out which endpoint to call. Perhaps that is why Swagger (also known as Open API) has steadily increased in popularity over the past five years.

With GraphQL, you have a single endpoint. There are queries, mutations, and a schema. Queries allow you to write little mini-programs you send to that single endpoint that fetch data in the context of the schema. Mutations, however, are just straight up RPCs.

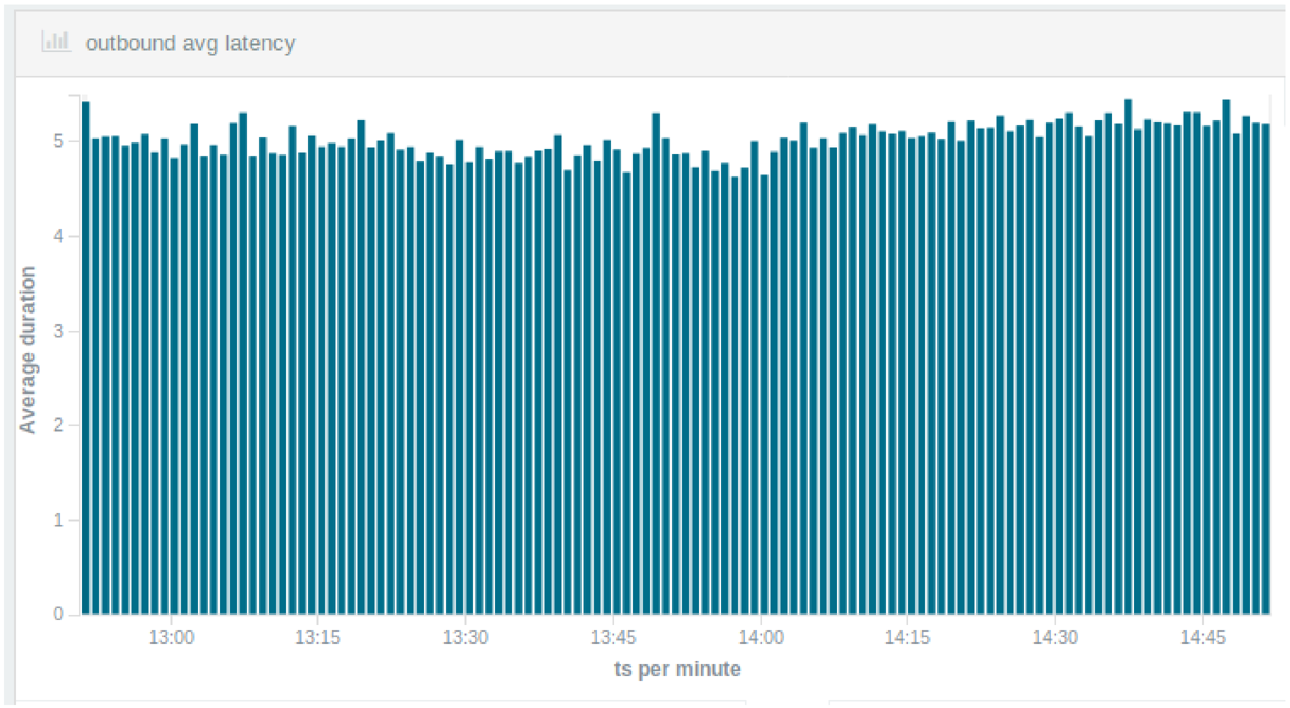
Design

What are the major differences between the technology designs of these two microservices?

The API for feed 4 is based on the same Swagger specification as the feed 3 - 9 implementations. This implementation uses a Node package called swagger-tools which reads the Swagger spec then generates the routing for that spec within the Express framework. That routing maps each request to the corresponding controller which, in turn, invokes the appropriate service. There are usually two handlers for each entity. One for GET and one for POST. The outbound entity has a third handler for keyword based search. The feed 4 implementation uses the low level drivers for Cassandra, MySql, and Redis. It uses the Node http module for accessing Elasticsearch.



In feed 10, the schema, queries, and the mutations are specified as a set of type definitions. The connections to the underlying datastores are all opened and used to initialize the services for participants, friends, inbound, and outbound. In the resolvers, each field in the schema and all of the mutations are mapped to their respective service calls. The GraphQL server is then initialized with these type definitions and resolvers then started in order for the service to begin listening on the configured port. Instead of the low level MySql driver, the feed 10 service uses TypeORM where annotated TypeScript classes map relational database tables to and from objects. The actual GraphQL server itself comes from the graphql-yoga project. The other viable alternative for Node is the Apollo server which is more popular. I chose the graphql-yoga project because it wraps the Apollo server in a way that is simpler to comprehend and easier to use.

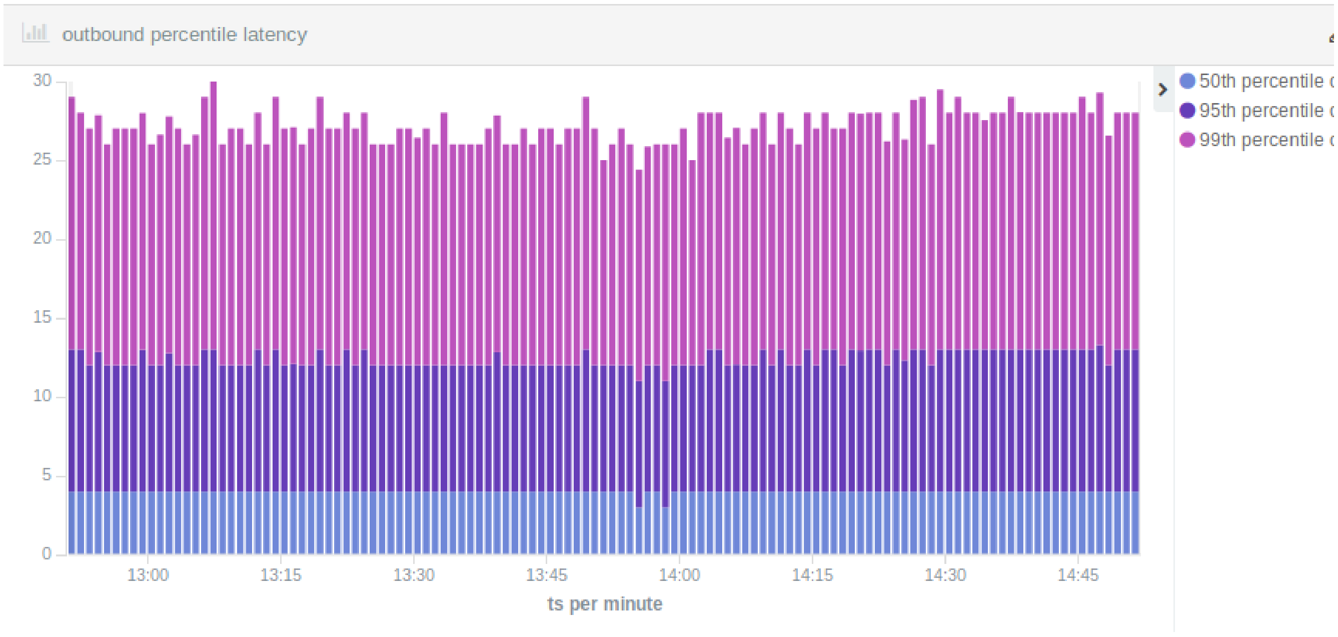


Code

What was learned in the actual coding of these two microservices?

Let’s start with some rudimentary static code analysis. In the feed 4 microservice, there are a total of 634 Lines of Code with a per file average of 39 LoC. In the feed 10 microservice, there are a total of 422 LoC with a per file average of 42 LoC. Cyclomatic complexity for feed 4 is 189 and for feed 10 is 672. I could not find a working package that computed the cyclomatic complexity for TypeScript so I first transpiled each file into JavaScript then computed the cyclomatic complexity with the same tool that I used for feed 4.

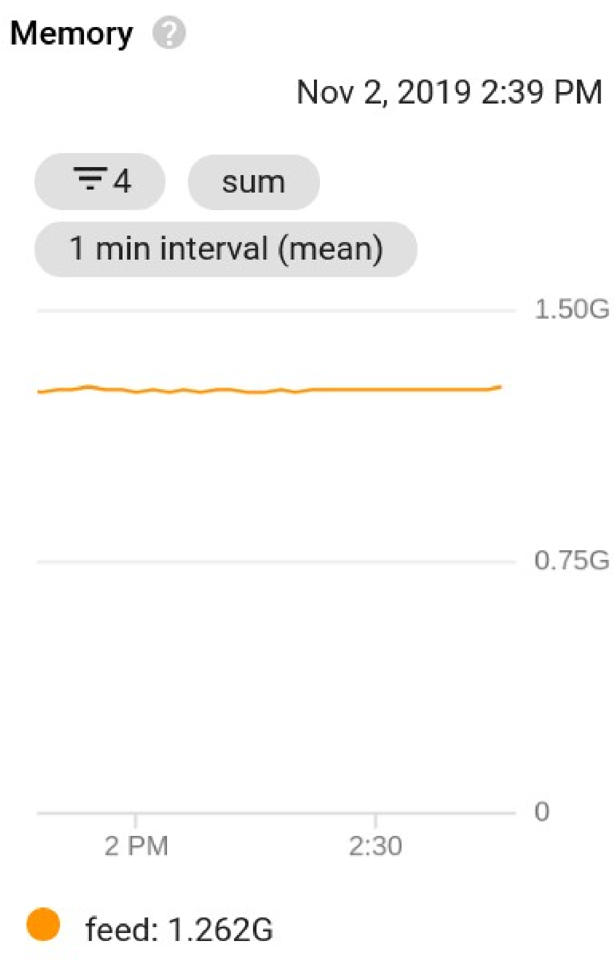
I was not successfully able to create a working unit test for this service. I tried many different test frameworks and mocking libraries. The trouble is with transpiling the service code that uses the Redis client. That driver is old school with callbacks. I use a package, called then-redis, which wraps the Redis client in promises such that it can be called by the async and await mechanism. This works perfectly in the service itself but confuses the transpile for the unit test run which complains that the return value for redis.get method should be boolean instead of a string.



Could I have found a way around this problem? Sure. I could have written my own Redis wrapper in such a way as to completely hide the actual Redis client. Part of the evaluation to any technology choice includes the entire ecosystem of related components. Having to rewrite a low-level component in order to make something as fundamental as unit tests working is not a good sign in terms of software maturity.

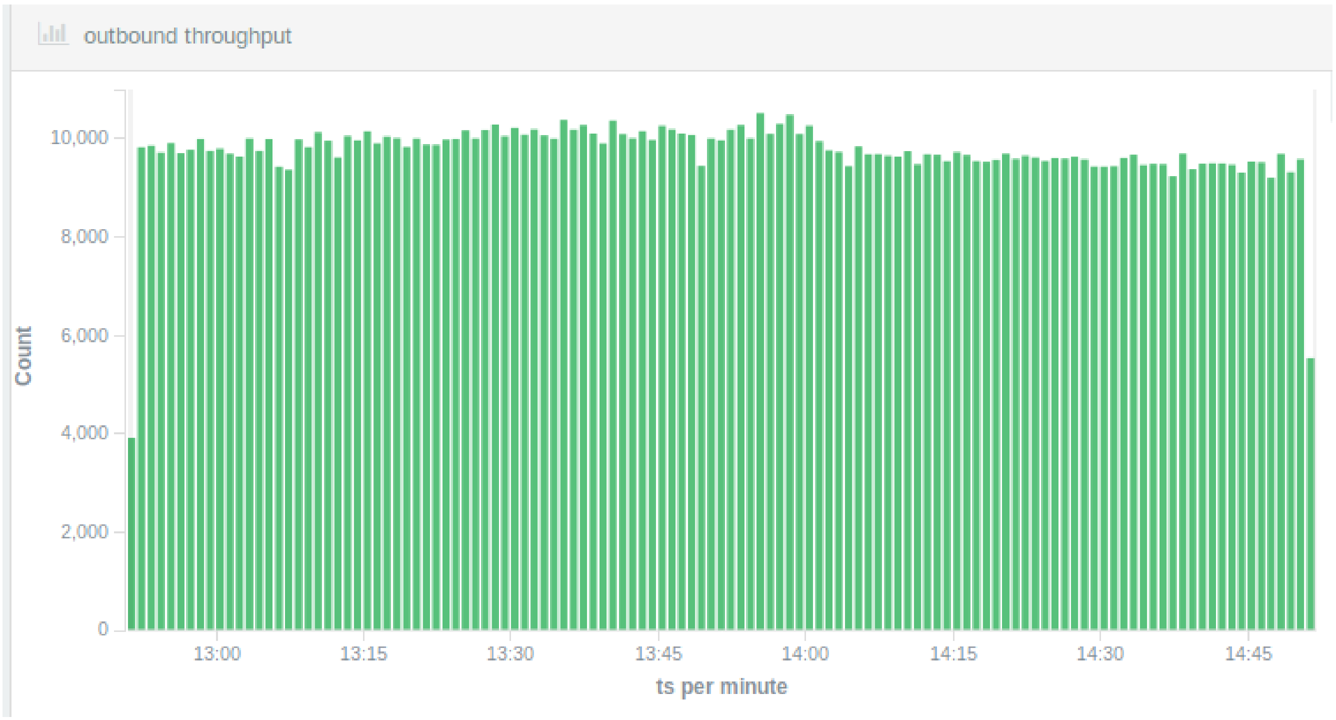
Performance

How did the GraphQL on TypeScript microservice compare to the Express on JavaScript microservice when it came to performance under load?



The usual performance analysis for these microservices focuses on capturing and analyzing the per minute throughput and latency of the create outbound call because that API does the most work. For feed 10, throughput was 9,817 RPM with a mean latency of 5 ms, a median latency of 4 ms, and a 99th percentile of 15 ms. For feed 4, throughput was 12,629 RPM with a mean latency of 5 ms, a median latency of 5 ms, and a 99th percentile of 12 ms.

I also wanted to analyze the performance for the create participant calls because that endpoint exercises MySql which is a big difference between the two implementations. The feed 10 microservice uses TypeORM instead of the low level MySql driver. For feed 10, average throughput was 3,874 RPM with an average latency of 10 ms. For feed 4, throughput was 4,995 RPM with an average latency of 7 ms.



The feed 10 service had profiling turned on during its load test. A significant amount of time was spent parsing the GraphQL requests and in the TypeORM module but not quite enough to completely account for the performance differences.

Conclusion

Which is better for microservice development in 2019? GraphQL or Express? TypeScript or JavaScript?

The GraphQL on TypeScript service was 22% less efficient than the Express on JavaScript service yet required a third less code to implement. Perhaps the former would be more compelling than the latter if you were willing to pay a bigger cloud bill in order to achieve a faster feature velocity. For real world applications, there is a lot of devops maturity needed for feature velocity so your results may vary.

Subjectively, I would say that the GraphQL on TypeScript code was less complex than the Express on JavaScript code. The framework oriented style of GraphQL may require a little getting used to by server-side JavaScript developers. Some of that complexity got pushed over to the applications that call the microservice because GraphQL APIs are harder to consume than RESTful APIs.

Before REST API design was like the wild, wild, West.

I prefer static typing which causes me to favor TypeScript over JavaScript but there are still some compelling maturity issues that need to be resolved before I would feel completely comfortable recommending TypeScript on Node right now. Part of the reason why feed 10 was less complex than feed 4 was because async and await is a lot easier to code than callbacks. Modern JavaScript that runs in Node can also use the async and await mechanism.

I feel like GraphQL queries could be a good fit for orchestration services. Also known as Backends for Frontends, these types of services don't access databases directly. Instead, BfFs orchestrate the calling of other microservices (known as data APIs) in order to fetch data. In that way, a GraphQL service could act as a simplifying facade over a complex collection of RESTful microservices.