Navigating Complex GraphQL Testing: Automating Integration Test Generation

Nate Smith, Ph.D.

nate@natesmith.io

Abstract

Care.com's federated GraphQL endpoint features over 450 operations, acting as a gateway to a multitude of backend data sources. Developing an additional layer of integration testing for our graph following internal staging deployments presented a significant challenge due to the vast number of possible operation variations.

Drawing on some work by Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark (Karlsson et al 2020), we built a script that automatically generates GraphQL API requests by traversing the graph under test, constructing a gql query, and then populating input variables using a property testing library. The result is an easy-to-run, self-updating test script that tests every operation with a variety of fields and input values without any need for manually writing GraphQL queries to test.

I describe the results of this effort, and then discuss what I think this says about what is possible with building integration tests for a complex system like this. While we cannot automate everything, sometimes it turns out that we might be able to automate more than we first think is possible.

Biography

Nate Smith is an accomplished software professional with over 15 years of diverse experience in the industry. He began his career in the early 2000s in software QA for medical devices. Shortly thereafter, he spent the next decade earning graduate degrees in philosophy, and then returning to the industry in 2011. Since then he spent several years working in quality assurance and building automated testing frameworks. He spent several years in engineering management across many different functions, including quality engineering, site reliability engineering, data engineering, and product localization. He is currently a principal software engineer at Care.com, where he has built several automated testing frameworks and now develops backend application systems.

Nate holds a PhD in philosophy from the University of California, Davis, an MSc in philosophy & history of science from the London School of Economics, and a BA in philosophy from UC Berkeley. During his academic career, he published multiple articles and presented his work at numerous national and international conferences. He specialized in the philosophy of science.

1 The Problem

Care.com's ecosystem of applications is supported by a federated GraphQL API containing more than 450 distinct operations. In 2022, we had numerous automated test pipelines testing various components of our systems including end-to-end tests for consumers of the GraphQL API, a robust unit test suite for the API itself, and unit and integration tests for the dozens of backend microservices supplying data sources. But we had no GraphQL-specific integration tests.

Integration tests at the API layer, especially if they are comprehensive, would be likely to uncover some kinds of bugs that would be unlikely to be detected elsewhere: integration failures between the graph and backend services, edge cases on GraphQL operations not current exercised by the consumers of the graph, and others. We decided to invest in a project to build out an integration test layer for the GraphQL API itself.

However, building even a basic set of integration tests making a series of GraphQL requests to cover our core operations would be an enormous undertaking, even if we distributed the work amongst our delivery teams. GraphQL requests are more varied and flexible than traditional REST requests, and to get adequate coverage across our hundreds of operations iterating over the many possible field choices and input parameters, we would need to build thousands of tests. When we get to this scale, it almost demands automating as much of this as possible to make it tractable, so we did some research into the possibilities. We found recent research into this problem by Karlsson, Čaušević, and Sundmark (2020) with a prototype and proof of concept for automatically generating GraphQL property tests and decided to operationalize this and bring it to our stack. Below I will describe the challenge in testing our graph in a bit more detail by contrasting it with testing a traditional REST API, and then describe our framework, what our results were, what we learned, and how we are moving forward.

1.1 Testing a REST API

A Representational State Transfer (REST) endpoint is a specific URI (Uniform Resource Identifier) that a RESTful web service exposes and through which the service's resources interact. This interaction is accomplished using standard HTTP methods, such as GET, POST, PUT, and DELETE.

When testing a REST API, we typically create a series of valid and invalid HTTP requests and send them to the targeted endpoint. These requests may include both required and optional fields. The complexity of these payloads can vary, ranging from single scalar values, to deeply nested JSON objects. But generally speaking, a good REST endpoint provides access to a single resource per request. We request a single user (or a list of users), but we if we want additional resources we make additional requests.

Let us consider a simple example for a GET request. Suppose we have an endpoint, "/users/{id}". This endpoint accepts GET requests, and has a single required value for the payload, *id*. Assuming for simplicity's sake that authorization is not required, if the user entity exists with that id, the server returns a response containing the user object. If it does not exist, we get back an error message indicating that the resource does not exist.

This user entity may be quite complex: it could have dozens of attributes, some of them nested entities themselves (for example, an address that has a number, street name, city, postal code, etc.). Once we know the structure of the possible response, testing such an endpoint is relatively straightforward. We might seed the server with data with various extreme values if we can, testing the boundary conditions. We may try sending various invalid inputs for the user id, but we are still just sending a single id to this endpoint.

Endpoints can of course support more complex payloads. POST and PUT requests in particular may have deeply nested JSON payloads including dozens of fields (perhaps we are submitting a web form). Such cases are more complex to test, but once we know the possible fields that will be accepted, we can

vary which ones we send, vary the data we send for each one, and verify that we get the expected responses. Furthermore, there plenty of tools in the community that can help generate and iterate on well-documented REST endpoints.

The general challenge of testing a REST API endpoint is to understand the possible input payload, possible response, and iterate on these. We can use tools to iterate on the inputs, and assert on the responses.

1.2 Distinct Challenges of Testing a GraphQL API

GraphQL operations come in two types: *queries*, requesting data, and *mutations,* modifying data. I will primarily be concerned with queries in this discussion, and in our test framework we restricted ourselves to testing queries, but in principle we could test mutations as well. They present similar challenges, but also have their own additional challenges of mutating data on test servers. For now, we are focused on queries. Let us stick with the *user* example.

First, note that GraphQL graphs are, of course, *graphs*. Nodes on the main graph are *fields*, information stored that has one or more relationships to other fields. Fields have types, either scalar (Booleans, strings, etc.), or objects, which are in turn defined on the graph. For example, a *getUser* operation might return a top-level *user* object, defined to have an *address* (among other fields), which is itself a defined object type. An *address* field might itself have a *street number* (a scalar string most likely), and so on.

The idea is that when we send a request to a GraphQL endpoint, we request only those values (‘fields’) that we actually need. With a REST request for a user resource, perhaps there is a lot of information that can come back on the response payload: user information, addresses, emails, orderIds, or whatever the business context requires. For a GraphQL request, however, my client might need a single field: perhaps I simply need to know the list of orders associated with a user, so I can specify that in my request. On the other hand, in some cases, the number of resources requested by a single request can be literally infinite in a sense (there’s a chance for infinite graph traversal loops in some graphs, but more on that later; that will need to be managed.)

Even if we restrict ourselves to a single operation, it is easy to start seeing why each operation gets complicated to test. Which fields do we request when we write our tests? Just some of them or all possible ones? If not all, which ones? How many combinations of fields do we test? This does not even get to the input data itself, which has the all the same complexity that testing REST endpoints does in varying the payloads. When testing our graph, we found that it was quite easy to generate requests that requested *hundreds* of fields. There is a new dimension of data -- the fields that we request -- that generates an explosion of possibilities. We need a way to create these, so we do not have to manually write them for all the operations. The challenge is overwhelming if we attempt to tackle it manually.

Even worse, there is no theoretical limit to how far the graph traversal goes. Some graphs have cycles (paths on the graph that form closed loops), and in such cases, the number of fields requested is literally infinite.

Given these challenges, we were going to have to either settle for a small set of manually written test scripts, or else find an automated way to generate them. In the next sections, we will delve into how we pursued the automated solution. It turns out that what seems to be a particular challenge ends up being a strength; we can exploit the graph structure of the API to generate test cases.

2 Prior Efforts

When researching possible ways to test our graph, we took note of a recent contribution from a paper titled “Automatic Property-testing of GraphQL APIs” published in 2020 by Karlsson, Čaušević, and Sundmark. They demonstrated that one could generate inputs for a GraphQL schema using a property-testing library, and generate the gql strings themselves (i.e., the payload that lists the fields requested) by

performing a graph traversal algorithm to build the strings. They were able to completely *generate* the test cases, managing both the field selection and input generation.

They built their proof of concept in Clojure, and to evaluate the efficacy of their solution, they injected faults into their graph and ran their automatic testing framework to detect these bugs. The results were promising - they identified 73% of the injected bugs. So, they are not catching everything, but they are finding *a lot*. Generating requests and using a property-testing framework to generate inputs and assert on them, resulted in finding many bugs. We decided to try this.

Care.com is primarily a TypeScript and Go shop, so we were not super excited about the idea of pulling in a new language / framework, learning it, and seeing how far we could run with it. Instead, we spent a couple weeks building out a few basic components of a TypeScript framework, that could handle a limited set of possible field operations. The initial version accepted query name, and built out a gql string to two levels deep for any query that had scalar inputs only (no objects), and then sent some canned values for the various input types. The initial results were promising (we found a few bugs immediately), and we continued handling all possible query types and more complex input objects.

3 On Property Testing

It is worth saying a little about property testing at this point; it ended up playing less of a role than we expected in our framework, but it is a core part of the idea.

At a high level, property testing is the process of varying the input to a system under test and asserting that the output conforms to a certain 'property'. This is different from another sort of test in which we validate behavior for specific input-output pairs, property tests articulate general properties of a system that should hold true for all inputs. Instead of “input A is expected to produce output B,” it is “the range of possible inputs of type X should produce the range of possible outputs of type Y.” To do this, we need two things: a way of generating input values, and a way of asserting on a range of output values. Rather than an expected output being, say, the *number 5*, we might expect that the output be a number in the range 1-50. We might know for a particular value that the system should only produce values in this range, so we generate a ton of inputs and make sure that no matter what we input, we still get an output in this range.

These input generators used in property testing are called 'arbitraries'. They generate a wide array of inputs, covering edge cases that a developer might not think to test. While this can include boundary conditions such as empty strings or very large numbers, the most valuable aspect of these arbitraries is their ability to generate truly random data. This randomness often leads to the discovery of unforeseen bugs and is a key reason behind the effectiveness of property testing.

When the property-testing library is generating values and asserting on the property, if a failure case is found, the library will try to isolate the range of values that produce the failed result by iterating over ranges of values and narrowing down to the failing range. This process is called “shrinking”: simplifying the input range as much as possible to identify the precise failure conditions. The library will continue to generate values and test outputs until it identifies this precise range.

In the context of our GraphQL API, the goal was to build arbitraries that could generate valid queries to test against our system. These queries would have to respect the structure of our GraphQL schema while introducing enough variation to uncover any potential issues; you will see how we do that below.

4 Approach and the Framework

Care.com’s GraphQL API is built with TypeScript; we decided to build the testing framework using the same stack. We wanted something familiar to our developers, and the tooling for client libraries in TypeScript is excellent given the extensive frontend GraphQL client ecosystems from organizations such as Apollo (https://www.apollographql.com) and Prisma (https://www.prisma.io/graphql).

We used FastCheck (https://github.com/dubzzz/fast-check) for the arbitraries; this is a property-testing library for JavaScript / TypeScript that is being actively maintained. We also used graphql-codegen for building a JSON representation of our graph from the graph schema, enabling the automatic generation of queries for testing.

All of these pieces put us in a place where we could build an automatic test-case generator and runner, using a depth-first search graph traversal to build a payload of requested fields, and fast-check to generate values for the input values. We iterated over a few weeks, and ended up with the core of our framework that generated and ran test cases for all the queries on the graph.

The core logic of the test pipeline is as follows:

1. Schema Retrieval: To test the latest version of the graph and represent the graph in a form that our TypeScript code can interpret, we retrieve the latest version of the graph as a gql file. We then run graphql-codegen to transform the file into an introspection JSON; essentially a JSON representation of the schema. (If introspection is enabled on your graph, you may be able to retrieve such a JSON file directly from the graph.) This representation of the graph includes all the queries, mutations, fields, inputs, and their types.

2. **Generate List of Queries**: We load the introspection file, and then iterate over the array of query names in the QUERIES node to extract the list of all queries in the schema.

3. **Load Configuration including Users**: Here we load the configuration for the test run, which includes a max_depth, a field selection strategy (all vs random), a maximum arbitrary iteration number, and an array of users to iterate over.

Then, for each query to be tested:

4. **Authenticate against the graph:** We run through each operation with each user, so first we authenticate against the graph with the user credentials.

5. **Generate the GQL string**: Starting with the query name, run a depth-first search graph traversal to the maximum depth specified in the configuration and add each discovered field to the gql string. If we hit the maximum depth and we are at a node that has required fields, we remove fields until we get to a valid gql string. If this does not end up with a valid gql string, we throw an error.

6. **Generate the Input Object**: Construct and input object for the variables in the query. For each input variable, the system parses its type, and then matches the type to a predefined data generator driven by fast-check. Some of these data generators produce random strings, while others generate more specific data such as zip codes or phone numbers.

7. **Send the payload:** Send the request to the endpoint.

8. **Iterate on input**: FastCheck steps in here; if the response is success, continue generating inputs and making requests out to a maximum specified by our configuration. If there is a failure, start ‘shrinking’ – FastCheck iterates over requests trying to find the limits of the failure range.

8. **Assert that the response conforms to a known property**. At the most basic level, assert that the response is not an error object or error response. Fail the test if this is not the case. It would also be possible at this point to make a more complex property assertion. For example, we might wish to assert that if a given field requested is of type ‘zip code’, assert that the field returned is a five-digit integer. (We so far have not chosen to do this, as the server itself enforces its own schema validation on returned values and adding this to framework is a decent amount of work. But it would be a prudent extra verification.)

9. **Test Completion**: The testing framework concludes the test run by compiling a report of the failures. It leverages fast-check's property test reporting to provide detailed information on the values and ranges

that caused each query to fail. This report offers valuable insights for troubleshooting and improving the GraphQL API.

In our approach, we should note that our framework, at this stage, largely uses arbitrary or random inputs for test queries. The fast-check library, which we employ to generate these inputs, is not equipped with the inherent knowledge of the business logic associated with a particular field. Hence, unless we specifically define this logic ourselves, it generates random data inputs like strings, integers, Booleans, etc.

However, we have tried to construct more intelligent generators for certain fields to ensure they generate valid data according to our schema. For instance, for fields like phone numbers, zip codes, and user UUIDs, we have built generators that produce valid US phone numbers, valid zip codes, and UUIDs referencing actual users, respectively.

This approach highlights one of the framework's limitations: without tailored generators, the data inputs can sometimes lack the necessary contextual relevance.

On the other hand, an important feature of this framework is its adaptability to changes in the graph schema. As the schema evolves, there is no need to update the code in the tests. The test cases are generated directly from the schema, allowing them to stay up-to-date without manual intervention. This flexibility becomes particularly significant if we aim to introduce more specific business logic into certain arbitrary generators for new queries.

Lastly, it is worth mentioning that the current framework tests only queries, not mutations. We chose this approach mainly for pragmatic reasons, given the potential issues of mutating data in rapid succession on our test servers. However, running queries was our initial objective, and we consider the extension of this framework to test mutations as the next step in our progression.

5 Results

Once the framework was built and the initial bugs worked out, the resulting system had several noteworthy features. First, we set up a scheduled job to run these tests twice a day, providing consistent and regular testing of the API. This job generates hundreds of input sets for each query, covering a wide range of possible requests.

On receiving responses, the system checks if the response corresponds to a known bug or an acceptable error — these are predefined in our fixtures. If the response does not fall into either of these categories, the system asserts that it receives a known GraphQL object defined in our schema.

The implementation of this framework led to the immediate discovery of a few dozen bugs that fell into a few different categories. First, there were plenty of edge cases discovered for extreme input values. The endpoint would throw an error for certain queries and input value combinations. Second, there were some queries that just were not being used and threw errors for all the inputs they were sent. Third, there were many queries throwing errors rather than returning structured error objects defined in the GraphQL schema.

Since the framework has been stabilized, the tests have been detecting on average one bug a month. These are usually associated with new operations or changes to backend services that our system interacts with, although in a few cases they have indeed been additional failures in handling extreme input values or failures in error handling logic.

6 Discussion

From the operation and results of this testing framework, we gleaned several insights about Care.com's GraphQL setup, and indeed, GraphQL operations in general. These insights are not exclusive to Care.com's graph but are broadly applicable to any GraphQL operation.

First, GraphQL query strings can be extremely lengthy, even when the maximum depth is limited to 10 levels. In our tests, some query strings exceeded 1500 lines.

Second, consistent error handling is a crucial component for the successful implementation of a testing framework like this. It allows the system to properly respond to and record issues that come up during the testing process.

Lastly, this framework proved invaluable in identifying unused operations within our API. By testing all operations and identifying those which were never used, we were able to queue up a considerable amount of work to remove dead code, improving the efficiency and maintainability of our system.

We continue to test with the framework. After we got through the initial batch of bugs when we built the framework, we continue to see benefits: early warning on operations that are introduced to the graph but not yet functional, inconsistency between the graph's representation of a value and a backend service. (The most common version of this is a backend service that enforces a value as required, but the graph does not.) They have flagged insufficient error handling, and operations that are unused and support backend services removed (essentially pointing to tech debt we are hanging onto).

It has also been enormously helpful that since the schema is something we retrieve and update every time we run the tests, and the test cases themselves are automatically generated, we do not have any test maintenance to do as far as the framework itself. The test maintenance we do have is handling known bugs in the graph, and further improving arbitrary generation.

Beyond that, the most interesting lesson from this effort was that it ended up being possible to automate a lot more than initially thought. We had visions of ourselves writing out endless gql statements, and we think it pays to stop and think: how is this system *structured*? What about it is special such that we can programmatically solve this testing challenge. GraphQL's graph representation can be powerful and efficient for its consumers, but it also opened up a space of automated solutions.

However, there certainly have been challenges in pursuing this test strategy as well.

First, our graph is not perfect and like nearly any software system, we carry several known bugs and improvements that we have not yet prioritized. Since the test framework is automatically generating its tests based on the schema alone, we have to maintain fixtures that list known error conditions (and thus don't fail the entire test run when we see them again), and one that lists operations we simply skip entirely; these are almost always defunct operations that are no longer used that just haven't been removed from the graph yet.

Second, we have had to find a balance between random and 'realistic' inputs when generating values.

Finally, we have not yet tackled testing mutations. The additional complexity of making thousands of data mutations per test run and managing the effects of that is something that we have not yet started. The framework would work fine, we would have to make a few changes to support mutations, but we still have some thinking to do on what effect making so many data changes would have on our test systems.

7 Future Directions

As we look to the future of this testing framework, we are looking at several different improvements and additions. The first one is expanding the framework to test mutations. Mutations tend to have more complex request objects since they are modifying data, but there is nothing in the framework that would prevent us from doing so. One hurdle is managing all the changing test data, but that is something we can work through. Another is input validation: while our requests generated would be technically valid from a schema perspective, there is additional business logic behind the mutations that are not represented in the schema. We would need to add additional handling for these.

This leads us to another thing, which is that the current set of input generators, the arbitraries, are relatively 'dumb'; they do not have any sensitivity to the business context in which they appear. For each case, we would need to write-in in this sensitivity. If we do not do this, we can end up hitting simple form validation type errors before hitting the backend logic that we want to test more robustly. This tends to be especially acute with mutations, but it can occur with any operation.

For example, suppose we were testing addresses. Our backend systems may know that a city, state, zip code combination needs to match; it might throw a validation error if we try to submit an address that has a zip code known to be in California, but we submit a state of Oregon. If we end up just spamming the endpoint with various iterations of invalid combinations like this, we certainly get a fuzz test of sorts of the mutation, we end up stopped at the first layer of validation for most input combinations. It is certainly possible to write in constraints into the framework to enforce rules like this, but those have be dealt with on a case-by-case basis. We do have such constrained arbitraries, but there is room for many more of them and it would likely get us deeper into the logic of the system we are testing.

Acknowledgements

Thank you to the Care.com engineering organization for supporting this work and discussing these ideas with me. I would particularly like to thank Emmanuel Pamintuan Jr. for some early discussion and encouraging me to pursue this project.

References

Karlsson, S., Čaušević, A., and Sundmark, D. 2020 "Automatic Property-based Testing of GraphQL APIs https://arxiv.org/abs/2012.07380, (submitted December 14, 2020).