RFC 306 APPROVED: GraphQL API rate limiting

Editor: rslade@sourcegraph.com
Status: APPROVED
Requested reviewers: Cloud Security Eric Fritz
Approvals: Joe Chen Eric Fritz, Erik Seliger
Team: Cloud

Problem

Our GraphQL API is not rate limited in any way. This allows malicious or erroneous clients to make an unlimited number of requests to our API which could bring it down.

There are a few issues we need to solve which are in scope for this RFC:

1. How to track rate limits across multiple API instances and service restarts.
2. How to provide rate limit status back to clients.
3. How to allocate requests for authenticated and unauthenticated users.
4. How to calculate the “cost” of an individual request given that resources required for a single GraphQL query can vary wildly.
5. How to configure rate limits

Out of scope:

1. Non GraphQL endpoints will not be rate limited for now

The aim of this proposal is to get a first iteration of rate limiting out that is able to handle our goal of hitting ~1 million MAU.

Currently, we have about 100k MAU and which correlates with a peak of about 250 HTTP requests per second so we need to scale to approximately 2500 requests per second.

See this query which shows requests for the last 7 days.

Tracking rate limits

To track rate limit quotas across multiple instances we require the state to be stored in a central location which is accessible to all instances. The obvious choice here is Redis given it’s capacity for high read / write volume.

We need to decide which rate limiting algorithm to use. After some research it appears that the best option for our needs is the GCRA algorithm as it doesn't require a separate process to "fill the bucket". This article gives a good explanation on why it is better than a normal "leaky bucket" algorithm and also better than the naive approach used by GitHub:
https://brandur.org/rate-limiting

As a bonus there is an existing Go / Redis implementation that on inspection looks high quality and is apparently in production use at Stripe:
https://github.com/throttled/throttled

Specifically, it has a redigo store implementation:
https://github.com/throttled/throttled/tree/master/store/redigostore

We currently have two Redis stores, "cache" and "session". I think it's safe to use the cache store since we don't need to persist rate limits for a long period of time. The algorithm will adjust quickly even after losing it's state.

Providing feedback to clients

We'll respond with the standard rate limit HTTP headers as proposed here:
https://tools.ietf.org/id/draft-polli-ratelimit-headers-00.html#rfc.section.3

```
RateLimit-Limit
RateLimit-Remaining
RateLimit-Reset
Retry-After
```

Again, the "throttled" package above has us covered here although it prepends `X-` which we don't want:
https://pkg.go.dev/github.com/throttled/throttled/v2#HTTPRateLimiter

If the rate limit has been exceeded we'll respond with HTTP 429 (Too many requests)

Allocating requests

Requests on our internal endpoints will not be rate limited.
Requests with internal actors will also not be rate limited.

For authenticated users, requests will be allocated per user.

Unauthenticated users will be limited by IP address.

Protecting ourselves from large scale DOS attacks should be done prior to the application layer and we currently have Cloudflare in place.

GraphQL costs estimates

The impact of a single GraphQL query can vary. We therefore need a way to estimate the cost so that we don't limit simply on request counts, but limit on the cost of each query.

GitHub does this by estimating the cost. The gist is that they walk the query adding up all the "first" and "last" parameters as rough estimate to the number of objects that will be returned. The details are described here:
https://docs.github.com/en/graphql/overview/resource-limitations#rate-limit

We should analyse the request so that we can estimate the cost **before** running the query, We already have a naive version of this here:
https://github.com/sourcegraph/sourcegraph/blob/ce2b4c05689fa0d6295d4a5216f705a0d886da7d/internal/extsvc/github/v4.go#L169

This approach allows us to estimate the cost before performing any expensive operations or traversing our actual resolver graph.

Once we have an estimate we can check this against the rate limiter and decide whether to continue or not. A rejected request does not cost anything as we assume that performing an estimate calculation is cheap.

I propose we instrument all our GraphQL requests to get a feel for the range of this cost so that we can tune the algorithm and pick an appropriate value. We need to ensure that normal usage of our app does not cause us to hit any rate limit throttling. We should tag the costs by authenticated and unauthenticated users to see if there's a difference.

We can add a custom GraphQL directive as a hint for which queries or mutations are more expensive. By default everything will have a cost of 1 if no directive is added. Directives can be defined like this:

```
directive @cost(
   complexity: Float
) on QUERY | MUTATION

# you can define a cost directive on a query
type Query {
    getThings @cost(complexity: 2.0){
        first: Int
    }: ThingConnection!
}
```

GitHub also applies a hard limit to the number of nodes that can be returned. Perhaps we should add this too. An alternative GraphQL library, https://gqlgen.com/reference/complexity/, already supports limiting the complexity of a query however it is beyond the scope of this RFC to assess the complexity of switching libraries.

Configuration

We need to configure our rate limits and be able to disable them completely as it's likely that most customers won't require it.

The key points below are:

- Only authenticated users are rate limited
- Ability to override specific "keys". This could end up being a user, ip address or any other way of grouping requests which we may decide on later
- Rate limiting can be disabled site wide (the default) or per key.

Configuration will exist in site config and we will listen for updates:

```
"apiRateLimit": {
 "description": "Configuration for API rate limiting",
 "type": "object",
 "properties": {
   "enabled": {
     "type": "boolean",
     "default": false,
     "description": "Whether API rate limiting is enabled"
   },
   "perUser": {
     "description": "Limit granted per user per hour",
     "type": "integer",
     "minimum": 1,
     "default": 5000
   },
   "perIP": {
     "description": "Limit granted per IP per hour, Only applied to
anonymous users.",
     "type": "integer",
     "minimum": 1,
     "default": 5000
   },
   "overrides": {
     "description": "An array of rate limit overrides",
```

```
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "key": {
            "description": "The key that we want to override for
example a username",
            "type": "string",
            "minLength": 1
          },
          "limit": {
            "description": "The limit per hour. 0 means block all
requests",
            "type": "integer",
            "minimum": 0
          },
          "unlimited": {
            "description": "Set true to disable limiting",
            "type": "boolean",
            "default": false
          }
        }
      }
    }
  }
},
```

Supporting future use cases

In future we'll need to protect the search API as well as other shared resources for example executor jobs.

We can potentially use specialised keys to use a different cost bucket which would be trivial to implement.

Another option is that we continue to share the key, for example user_id and assign a relative cost to other requests.

This proposal I believe does give us enough flexibility given what we currently know about future requirements.

Definition of success

We are able to limit the rate at which clients can hit our API and we take into account the fact that GraphQL queries do not all incur the same cost.

We can easily disable the feature for customers who don't want it.

Limits can be updated in real time so that we can respond to potential threats if needed.

We'll need a documentation page describing how we calculate limits as well as details about response headers etc.

Default rate limits should be chosen so that "normal" usage by users of our web app will not trigger any throttling.

TODO: What about exhaustive search?

Feedback

Changes since the first round of reviews:

- Added a note about defining custom cost directives which we can use as hints to estimate the cost of a query or mutation.
- Added a section to definitions of success about ensuring that we don't throttle "normal" usage of our webapp.
- Added information about our current scale and where we hope to be in about a year
- We don't need to worry about unauthenticated requests to the API since we require auth
    - Config settings for this were removed
- We may want to add an in memory rate limiter in front of our auth check
- Mentioned that it's safe to use our Redis "cache" instance
- Added the "supporting future use cases" section