TECHNIQUES TO PROTECT YOUR GraphQL API

Benjie Gillam, GraphQL TSC


FAST AND SECURE
PROTECT YOUR API WITH
★ TRUSTED ★
DOCUMENTS
IF YOU CAN, YOU SHOULD
COMPATIBLE WITH
MOST GRAPHQL CLIENTS AND SERVERS

“Skippy” the frog: read later!

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

- Nulla quis semper diam
- Vivamus laoreet turpis eu euismod semper
- In non tempus arcu
- Nullam dapibus diam tincidunt egestas vehicula

Nunc vitae arcu eu lacus iaculis fermentum.

Maecenas tempor, mi vel posuere bibendum, tellus ipsum iaculis lacus, a fringilla orci nisl euismod tellus.

Integer egestas dolor eget suscipit aliquam.

Nunc vitae arcu eu lacus iaculis fermentum.

In hac habitasse platea dictumst. Nam commodo neque in tellus molestie commodo. Praesent id nibh ut nunc pretium semper eu non neque. Morbi auctor massa libero, sed luctus odio efficitur eu. Phasellus purus nunc, ultrices non libero sit amet, porttitor interdum lorem. Nullam lorem sapien, pretium et purus in, scelerisque hendrerit sapien. Mauris ac justo

Slides

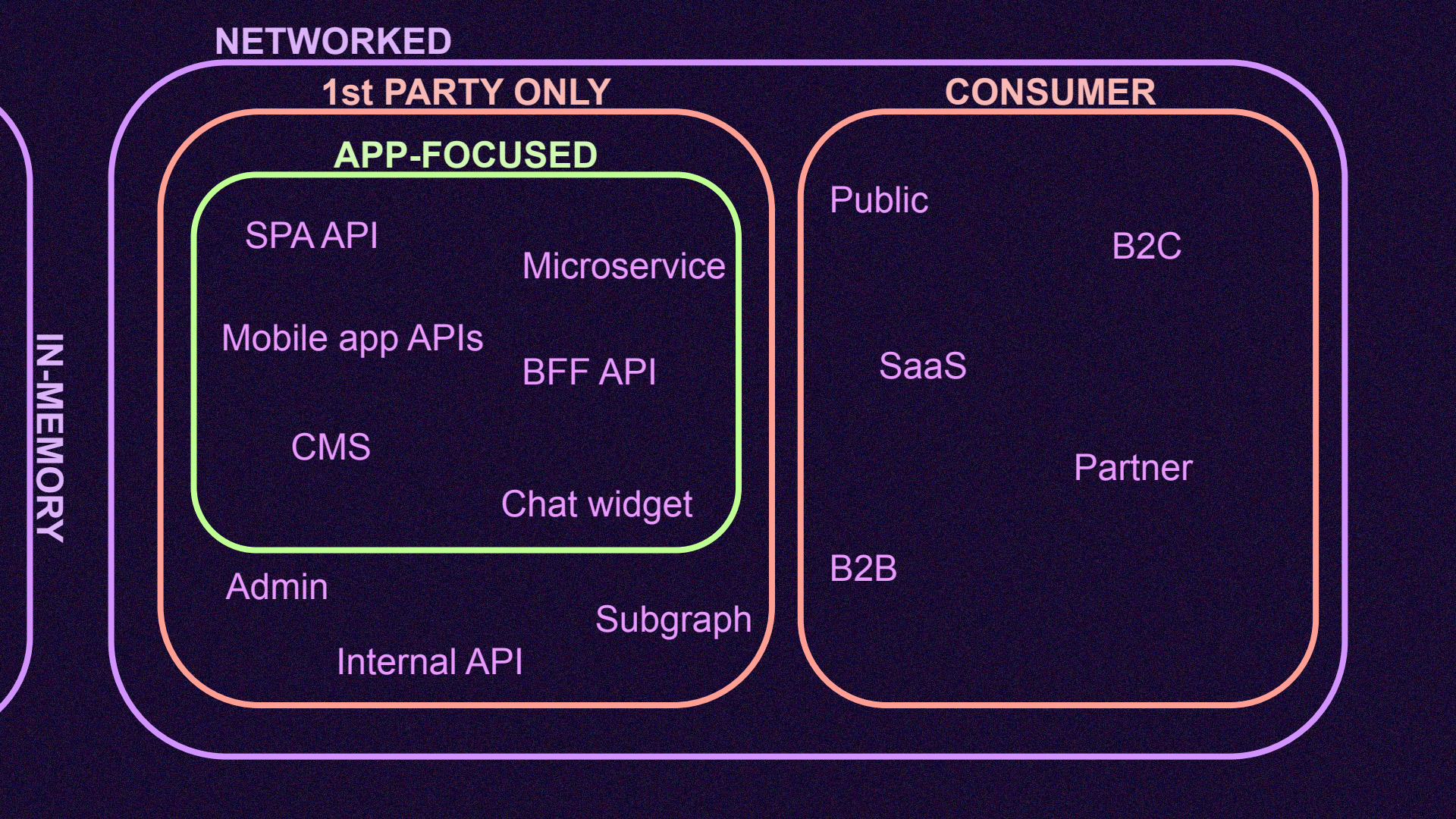
GraphQL as a function call

```
({ schema, operation, variables }) → { data, errors }
```

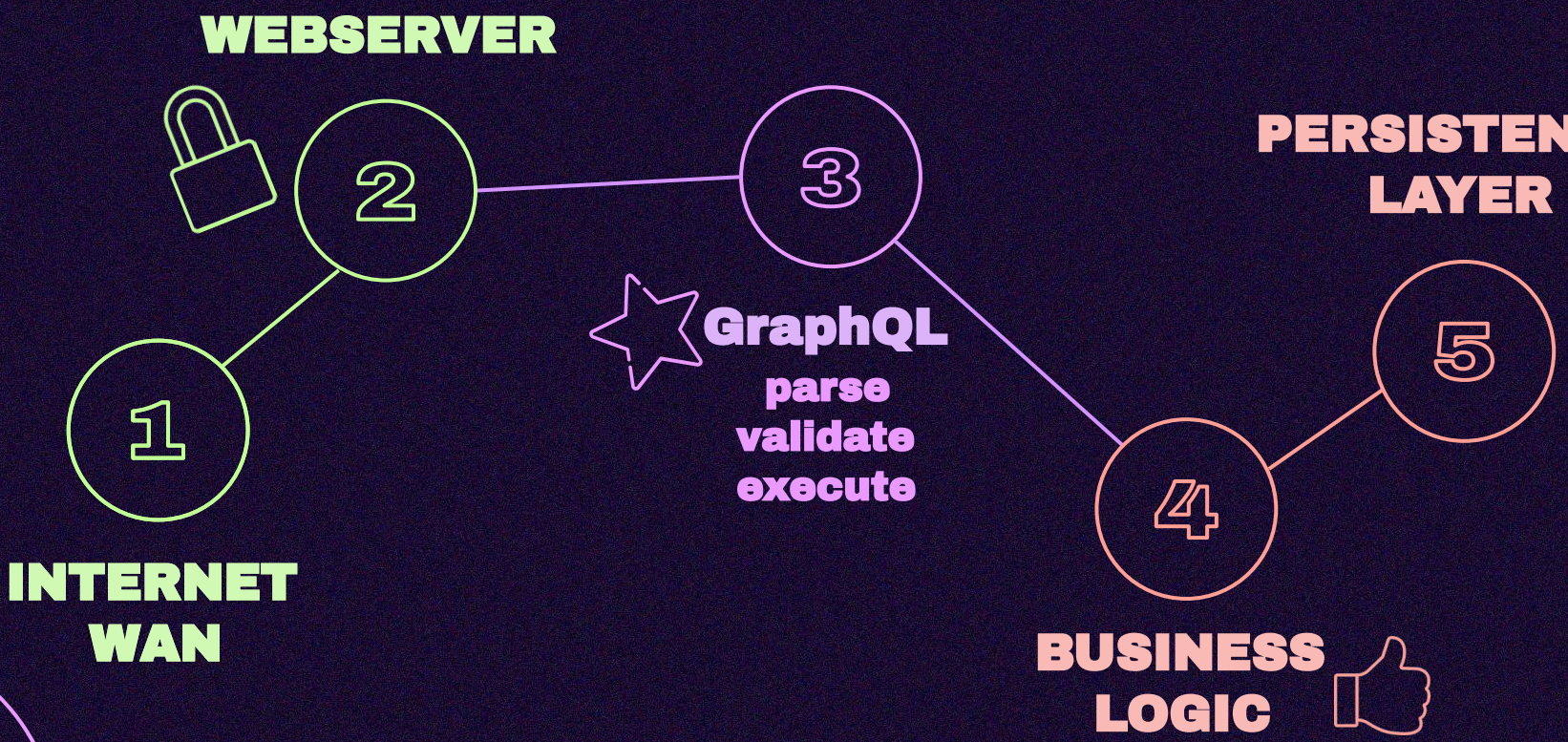
In-memory API concerns

Consumed only by your own application code in a privileged context.
Regular application code concerns:

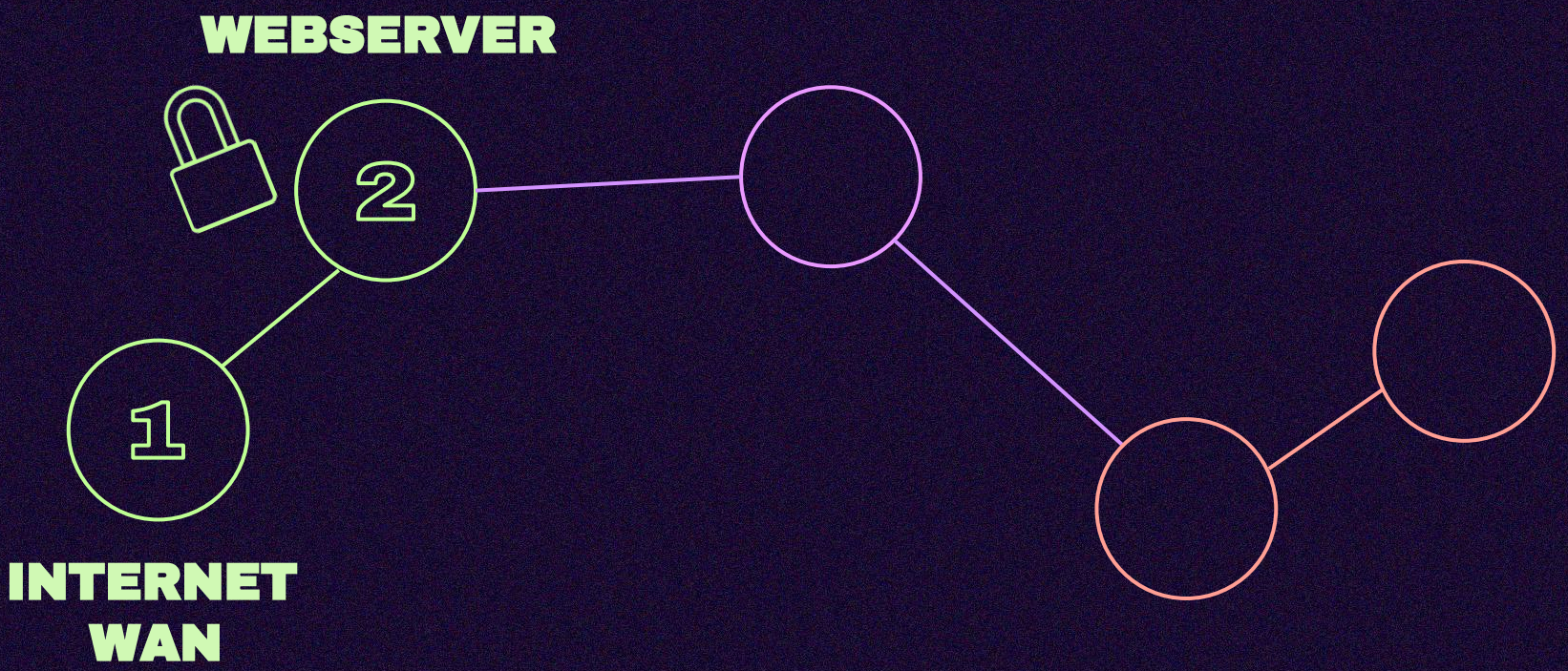
- Limit access paths
- Pass through authorization context
- Auditing
- Caching
- Be careful handling external APIs (e.g. error messages)
- Apply rate limiting / brute-force protection
- Consider circuit breakers


NETWORKED
1st PARTY ONLY
CONSUMER
APP-FOCUSED
SPA API
Microservice
Mobile app APIs
BFF API
CMS
Chat widget
Admin
Subgraph
Internal API
Public
B2C
SaaS
Partner
B2B
IN-MEMORY

NETWORKED APIs



HTTP CONCERNS



General HTTP concerns

Firewalls / private subnets / VPC / IP allowlisting
HTTPS Encryption
Request origin validation / CSRF (`SameSite` cookies / `X-CSRF-Token`)
Can you trust the proxy? (beware `X-Forwarded-For` / etc. spoofing)
Authentication
- Require all requests are authenticated?
- Scoped access / routing
- Beware session fixation attacks/long expiration/insecure token storage/etc

Auditing/tracing (including `X-Request-ID`)
Limit request size
Limit request duration and ensure cancellation on timeout
Avoiding over- and under-fetching
Rate limiting
Circuit break patterns
Protect other endpoints on your API host to avoid clickjacking/XSS/CSRF/phishing/etc
Disable proxy caching of sensitive data (`Cache-Control` / `Pragma` / etc)

GraphQL HTTP concern: CSRF (cross-site request forgery)

`multipart/form-data` and `application/x-form-data-url-encoded` bypass CORS preflight requests.

`evil.com` can make a request using your cookies!

Forbid these media types
Or: use a custom header,
e.g. `GraphQL-Require-Preflight`

GraphQL HTTP concern: batch requests

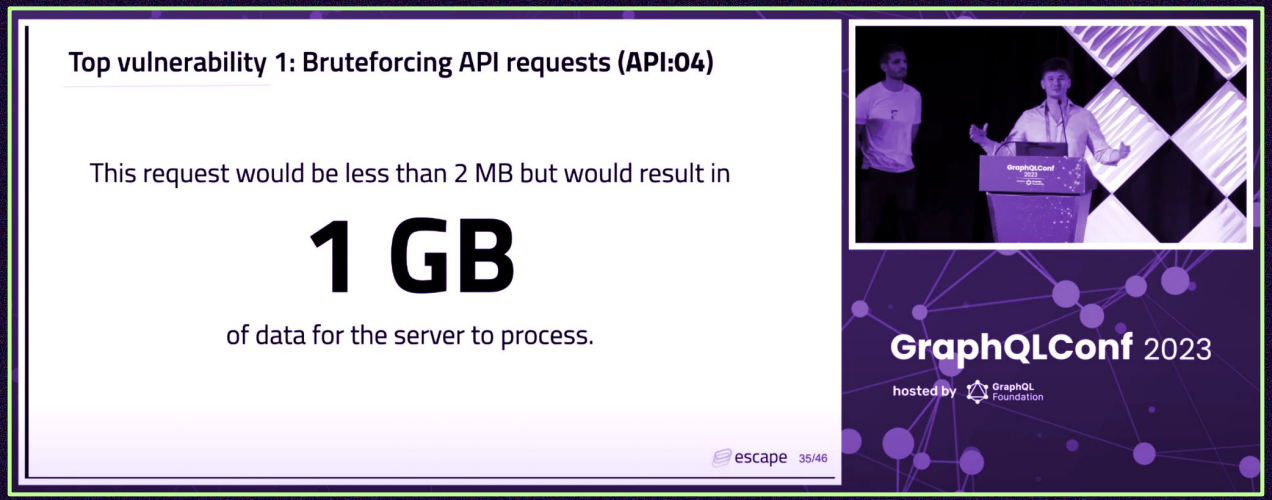
Circumvents HTTP-level rate limits.
Potential brute-force vulnerability.
Rate limits should factor in batch size.

```
[{"query":"mutation($u:String!){login(username:$u,pin:"0000")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0001")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0002")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0003")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0004")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0005")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0006")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0007")}"}
,{"query":"mutation($u:String!){login(username:$u,pin:"0008")}"}
```

GraphQL HTTP concern: GraphQL Bombs

Servers supporting `multipart/form-data` may be vulnerable to massive memory usage.

See “GraphQL Security Vulnerabilities in the Wild” by Escape.tech.



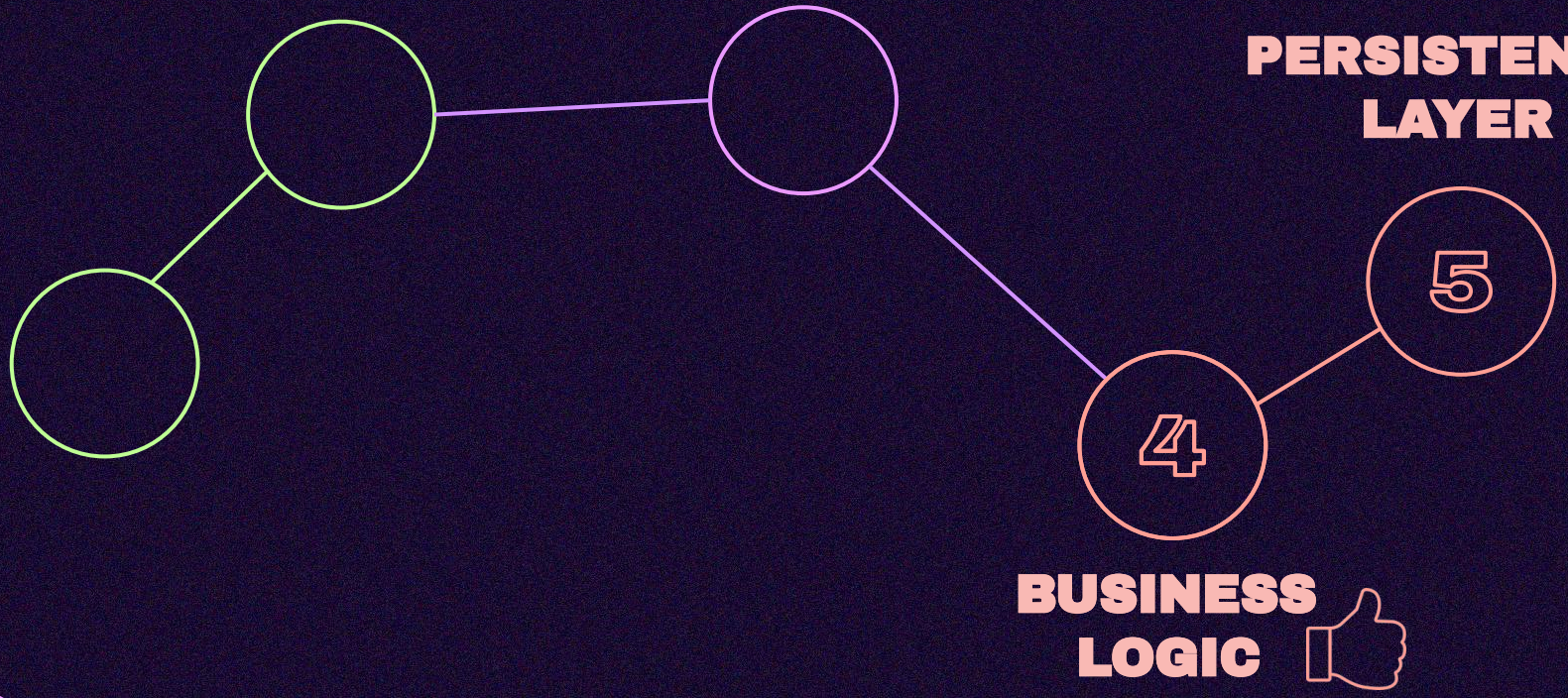
GraphQL HTTP concern: GraphQL Bombs

Either don't allow `multipart/form-data` or don't allow file uploads

Or **place limits** on the number of times a file-upload can be referenced:

- Webserver: don't allow batching when there are file uploads
- Schema design: don't allow file uploads anywhere inside of list fields
- Validation: don't allow a file upload variable to be referenced multiple times

BUSINESS LOGIC CONCERNS



General business logic concerns

Enforce authorization scopes (e.g. OAuth/etc)
Avoid injection attacks (e.g. SQL injection)
Implement input validation / sanitization
Where appropriate, use output masking to avoid leaking private data
Avoid remote code execution (RCE) vulnerabilities
Ensure dangerous operations produce an audit trail
Flag unfamiliar usage patterns / detect bad behaviors

- Track actor: which application? which user? `X-Request-ID`?
- Track error rates / request times / stability issues
- Pro-active blocking / use the breaker pattern (per-user, per-application, or globally)

Avoiding DOS: efficiency is key

Denial of Service (DOS) attacks often try and make servers perform disproportionately more work than clients. Ensuring servers are efficient helps make this more challenging to attack.

Use a **cancellation token** to cancel all executing business logic when a request terminates.

Avoid N+1 using batched execution

GraphQL's execution model, via resolvers, can lead to an explosion of requests to your business logic or persistence layers.

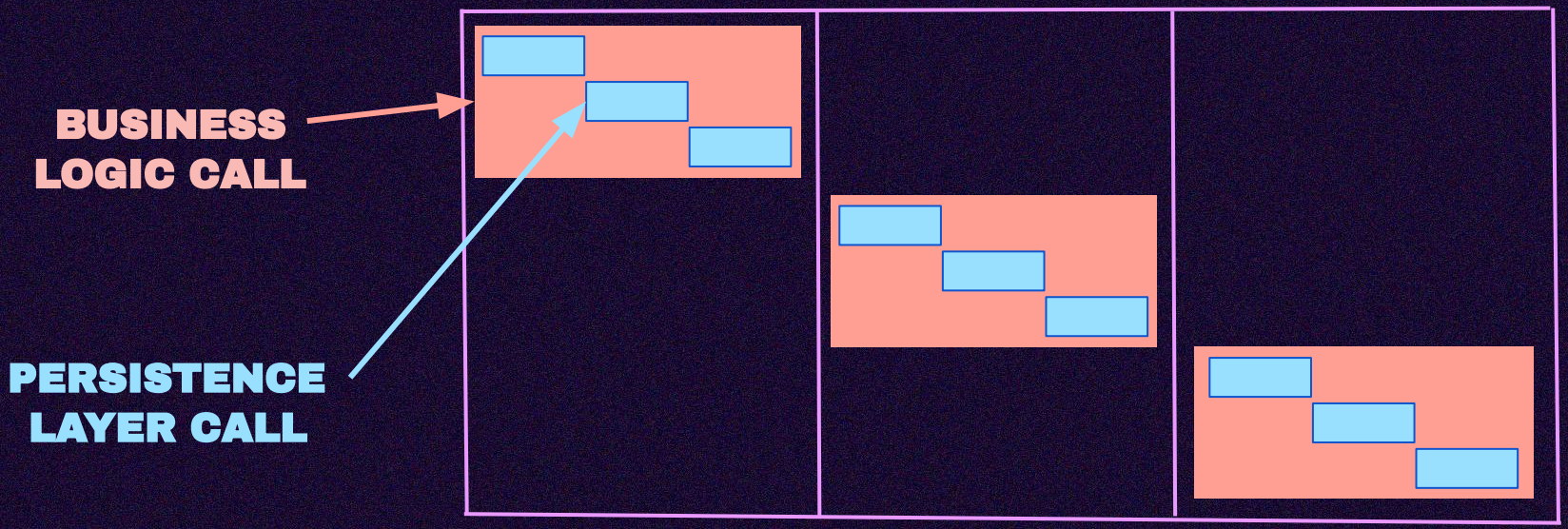
Use **batching** (e.g. via DataLoader) to turn hundreds of `userById(id)` calls into a single `usersByIds(ids)` batched call which can be executed more efficiently.

Advanced GraphQL implementations such as 

bake batching into the system's design so you never need to concern yourself with the N+1 problem.

Avoid over- and under-fetching server-side

Ensure your business logic is **optimized** to reduce round trips to the persistence layer.



Pagination concerns

Limit/offset pagination is fairly standard; but `offset:1000000` means a million rows must be executed before retrieving the rows you want.

Place reasonable limits on offsets.

Use **cursor pagination**; the cursor can be used as a predicate allowing the persistence layer to jump straight to the relevant records using indexes - no row-skipping required.

GraphQL LAYER

Handling Malicious Queries

Protect **app-focused** APIs with a **document allow list**

Use static queries

Dynamic query (string interpolation)

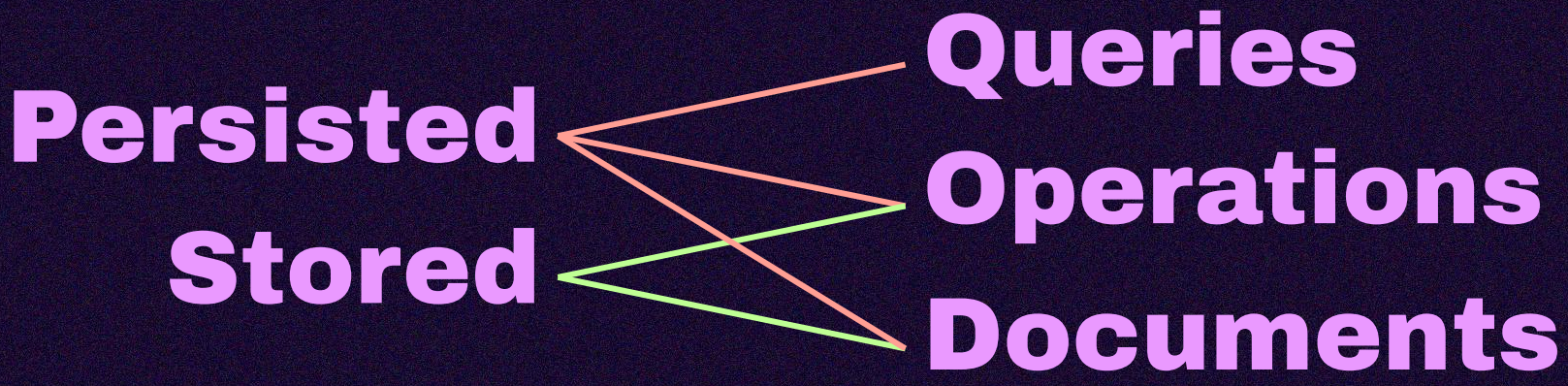
```
query = "query GetUser { user(id: "
        + userId
        + ") { name"
if (showAvatar) {
  query += " avatarUrl"
}
query += " } }"
```

Potentially infinite documents issued to server.
Server must validate each new document.
Hard to analyze/lint.
Cannot easily validate; needs thorough testing.
Vulnerable to GraphQL injection.

Static query (separate variables)

```
query = "
  query GetUser(
    $userId: Int!
    $showAvatar: Boolean! = false
  ) {
    user(id: $userId) {
      name
      avatarUrl @include(if: $showAvatar)
    }
  }
"
variables = { userId, showAvatar }
```

Reusable document, give variables at runtime.
Syntax highlighting, linting, auto-complete.
Not vulnerable to GraphQL injection.
Validate once, no need to re-check.
Supports "persisted queries" pattern.
Can check validity at build time.
Easy to track which fields are used.
Server can optimize known queries.

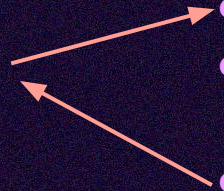

Persisted
Stored
Queries
Operations
Documents

Persisted queries

Client

Before client deploy:

- Extract all GraphQL operations
- Negotiate identifier with server
- Store identifier



Server

Before client deploy (example):

- Receive document from client
- Generate identifier, store id & document
- Return identifier to client

On GraphQL request send:

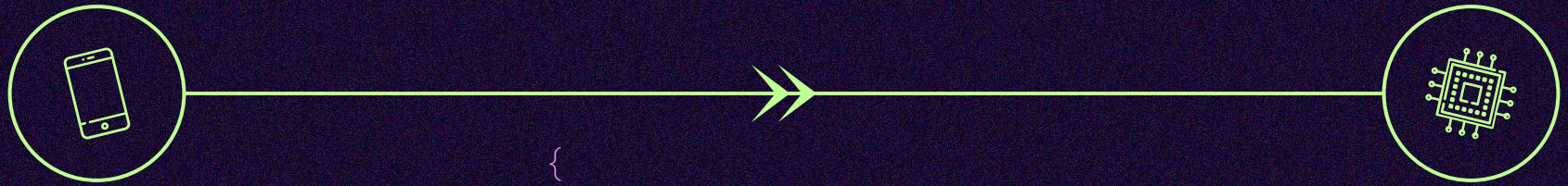
- Issue identifier to server, not document

On GraphQL request received:

- Look up document by identifier
- If no doc found, reject request (**allowlist**); or handle as desired
- Continue to execution as normal

Support

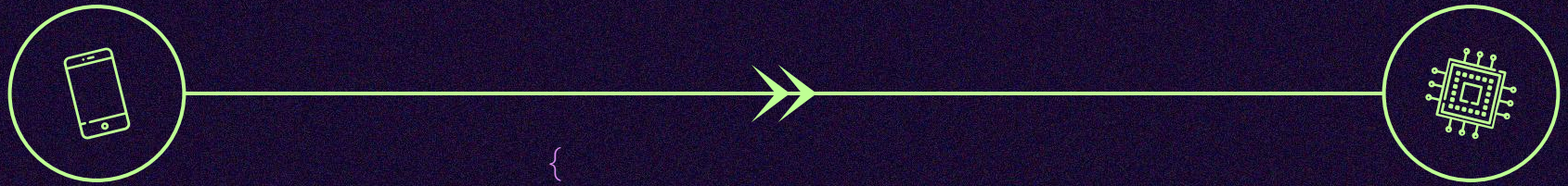
Wide-spread client and server support



```
{
  query: "query Q …",
  variableValues: {…},
  operationName: "Q"
}
```

Support

Wide-spread client and server support



```
{
  id: "sha256:…",
  variableValues: {…},
  operationName: "Q"
}
```

Support

Wide-spread client and server support

Persisted queries

- Can act as **document allow-list**
- Optimize network
- Validate at build time
- A little effort to set up
- Official specification (WIP)

- Custom URLs `/graphql/p/:hash/:name`
- Easier tooling integration
- Better caching
- Better tracing
- Easier debugging

Automatic persisted queries (APQ)

- ✗ Allow ***any*** GraphQL query


TRUSTED
DOCUMENTS

Trusted documents: persisted queries + trust

Trust the process:

- Trusted developer wrote the document
- Trusted review process (pull requests)
- Trusted CI checks (`gqlcheck` or similar)
- Trusted retrieval of document from server

If you can trust your documents, there's no need for expensive run-time validation.


FAST AND SECURE
PROTECT YOUR API WITH
★ TRUSTED ★
DOCUMENTS
IF YOU CAN, YOU SHOULD
COMPATIBLE WITH
MOST GRAPHQL CLIENTS AND SERVERS

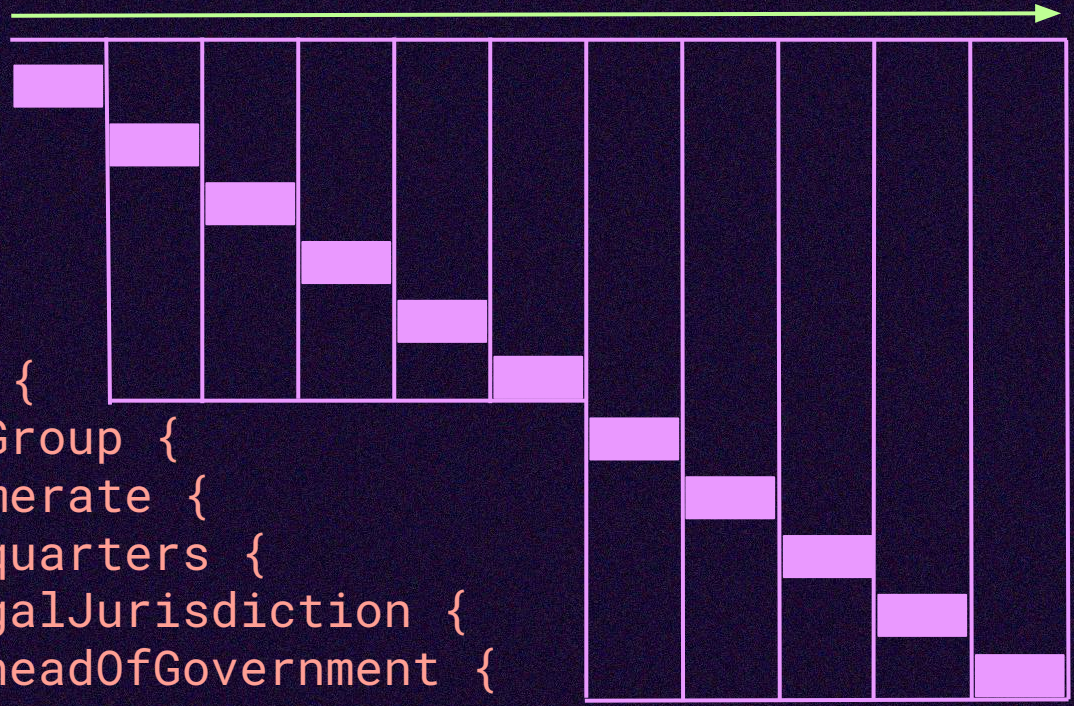
Handling Malicious Queries

Protect **consumer** APIs with **advanced validation**

(Also useful during development for any GraphQL API, even local!)

Depth limit

```
{
  item(id: $id) {
    position {
      shelf {
        aisle {
          store {
            localGroup {
              nationalGroup {
                conglomerate {
                  headquarters {
                    legalJurisdiction {
                      headOfGovernment {
                        name
}}}}}}}}}}
```



Request waterfall

List depth limit

```
{
  allFilms(first: 100) {
    cast {
      children {
        films {
          crew {
            name
 }}}}}}
```

100

100 * 100

100 * 100 * 2

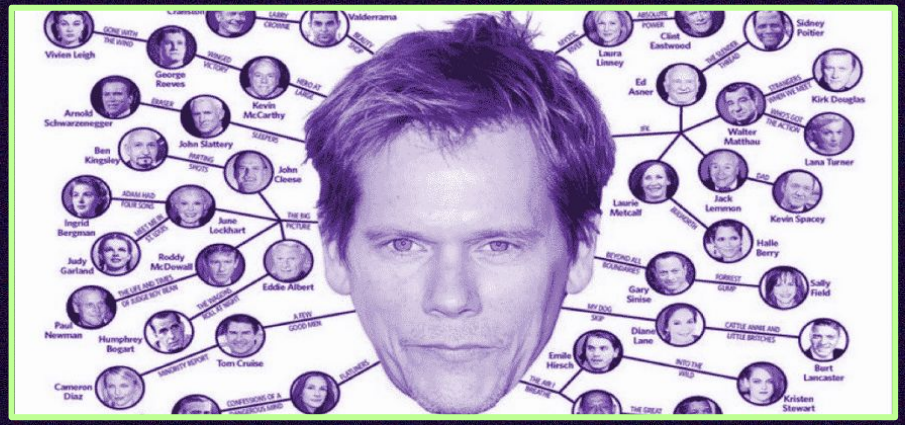
100 * 100 * 2 * 20

100 * 100 * 2 * 20 * 100

40m+ nodes evaluated

Self-referential limits

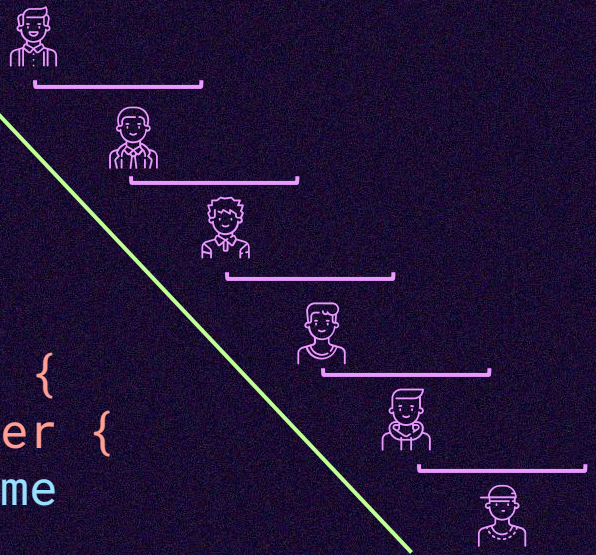
```
type Actor {
  id: ID!
  name: String!
  appearances: [Appearance!]
}

type Appearance {
  film: Film
  actor: Actor
}
```



Self-referential limits

genealogy.site

```
{
  father {
    father {
      father {
        father {
          father {
            father {
              name
}}}}}}}
```



friends.site

```
{
  friends {
    friends {
      friends {
        friends {
          friends {
            friends {
              name
}}}}}}
```

Introspection depth limits

Limits must be introspection-aware, or tooling may break.

Allow

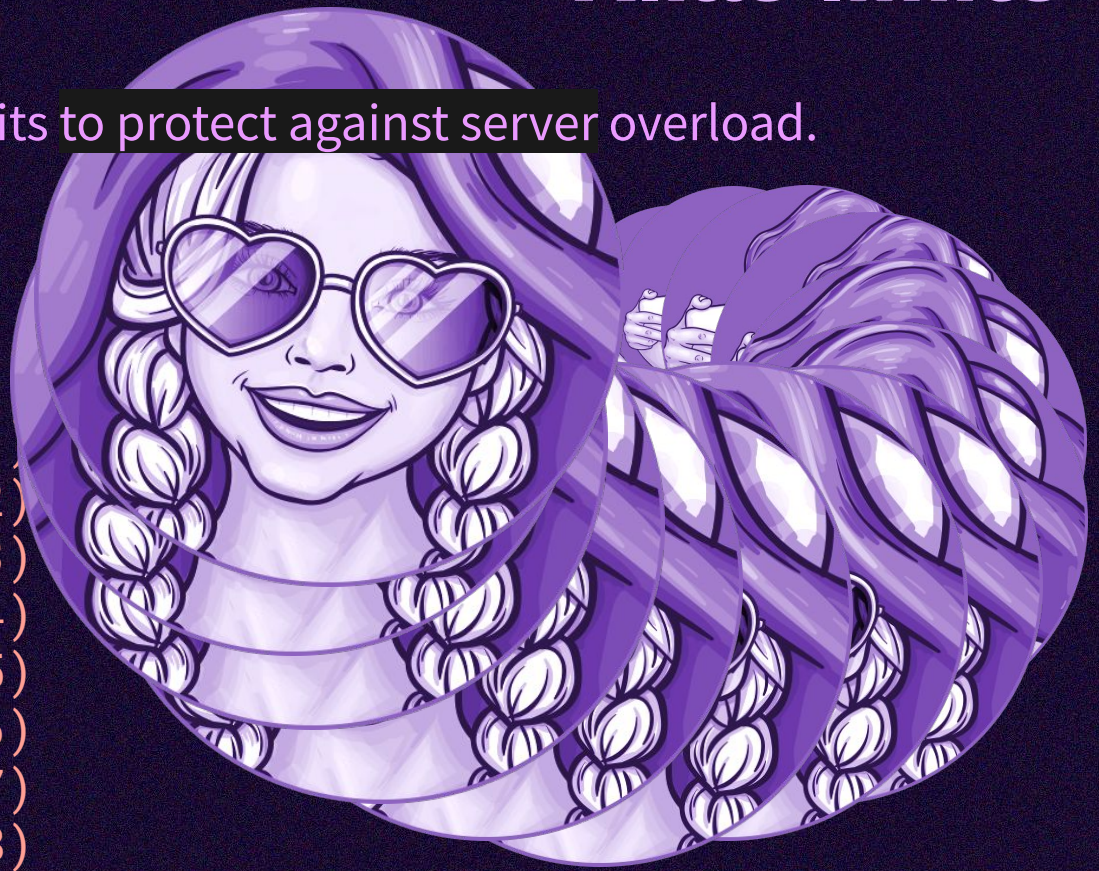
```
{
  ofType {
    ofType {
      ofType {
        …
```

Block

```
{
  interfaces {
    possibleTypes {
      interfaces {
        possibleTypes {
          interfaces {
            …
```

Alias limits

Apply sensible alias limits to protect against server overload.



```
{
  a1:avatar(size:1)
  a2:avatar(size:2)
  a3:avatar(size:3)
  a4:avatar(size:4)
  a5:avatar(size:5)
  a6:avatar(size:6)
  a7:avatar(size:7)
  a8:avatar(size:8)
  a9:avatar(size:9)
```

Alias limits

Attackers can bypass HTTP rate limits with brute force.

Protect from this in the **business logic layer**:

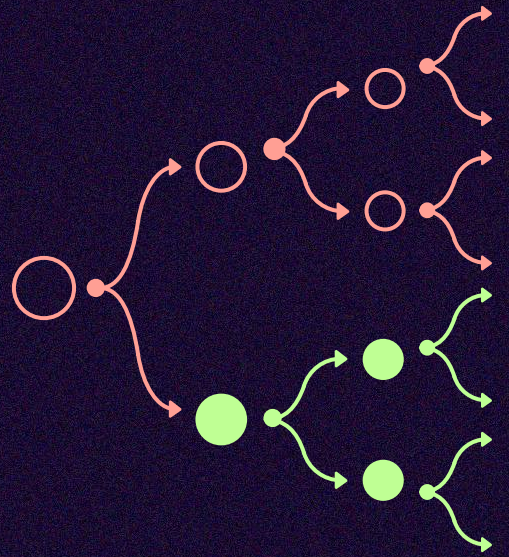
Use sensible limits on number of aliases; allow overriding on per-field basis.

```
mutation ($u: String!) {
  m0:login(username:$u,pin:"0000")
  m1:login(username:$u,pin:"0001")
  m2:login(username:$u,pin:"0002")
  m3:login(username:$u,pin:"0003")
  m4:login(username:$u,pin:"0004")
  m5:login(username:$u,pin:"0005")
  m6:login(username:$u,pin:"0006")
  m7:login(username:$u,pin:"0007")
```

Alias limits

Beware large selection sets even with low limit:

```
{
  u1: viewer { ... HugeFragment }
  u2: viewer { ... HugeFragment }
}

fragment HugeFragment on User {
  …
}
```



Custom Per-Document Validation

Depth limit
List depth limit
Self-referential limits
Introspection limits
Alias limits

Trust your devs,

or validate at development time.

Extra runtime validation rules required.

Migration cost when rules change.

Validation is an attack vector

Ten thousand spoons → ten thousand errors.

```
{ spoon spoon spoon spoon … spoon spoon spoon }
```

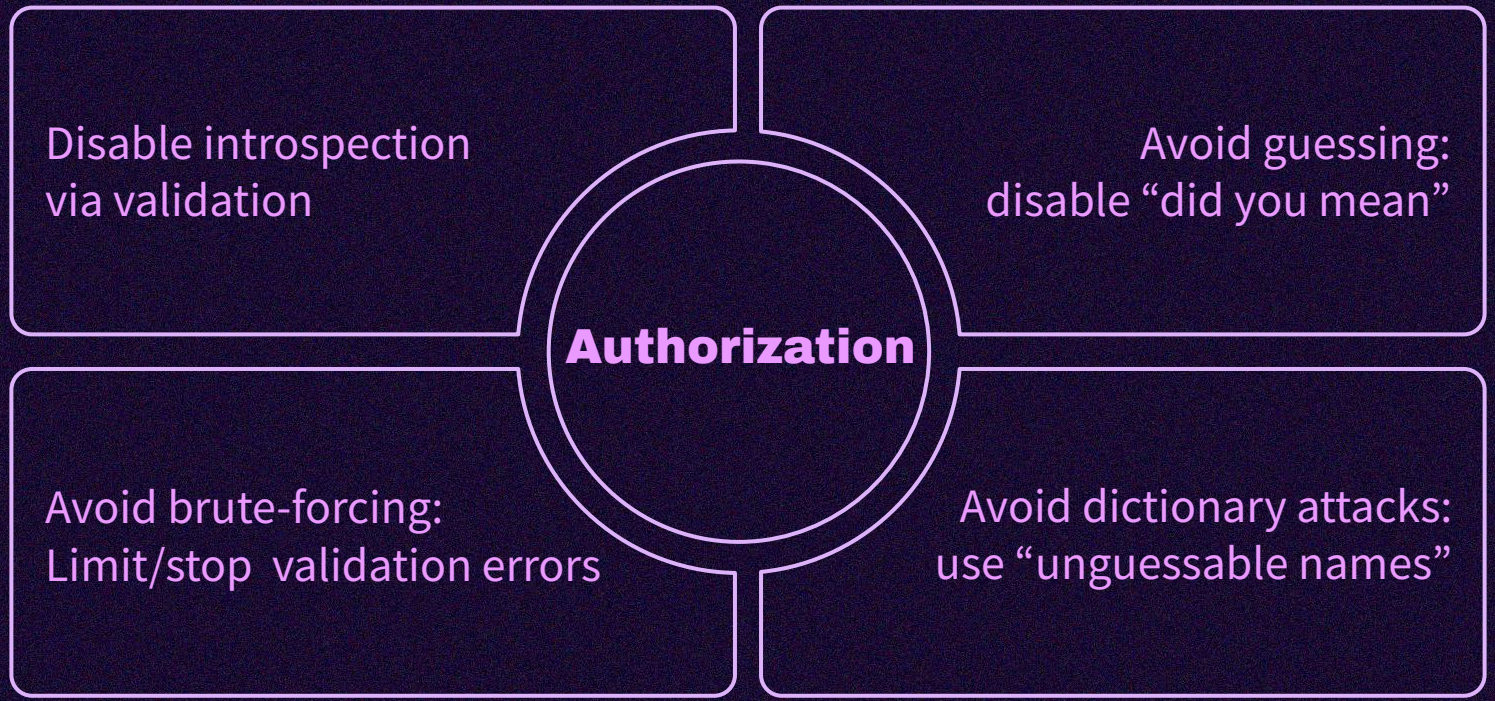
Limit number of errors from validation (e.g. max: 5)

```
{"errors":[{"message":
  "Too many validation errors, error limit reached.
Validation aborted."
}]}
```

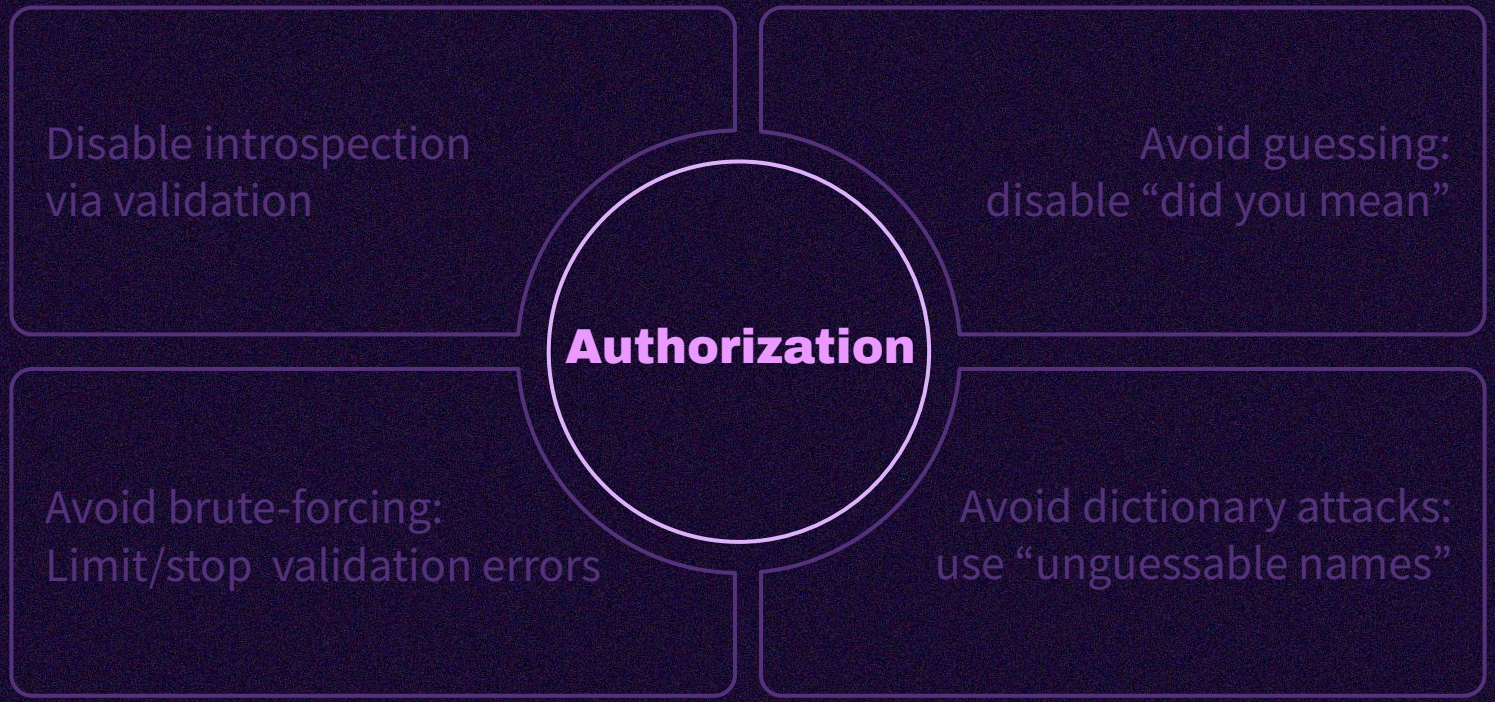
No action needed.

Validation needs to be able to abort once error limit is reached.

Prevent attackers from discovering your sensitive fields



Prevent attackers from executing your sensitive fields

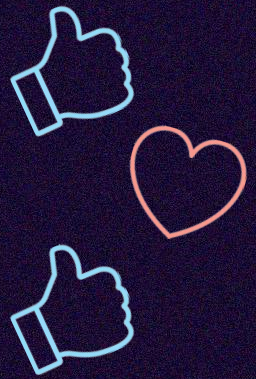


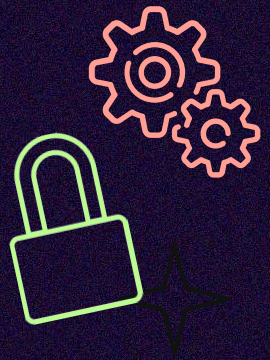
Schema design: avoiding information disclosure

Protect your privileged fields with authorization checks.

Only add **necessary and safe** fields to your schema.

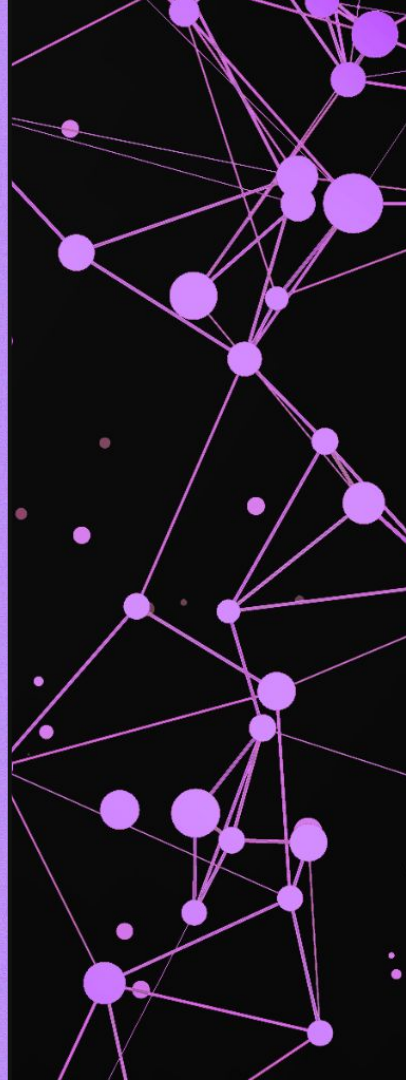
Consider using a separate schema for different use-cases (e.g. a separate Admin API).





Schema design: focus on efficiency

- use cursor pagination
- don't expose complex filters
  - only expose necessary filters
  - use simple arguments, not big filter objects
- enforce pagination limits
- avoid **`totalCount`** unless you need it



Disabling introspection: a red flag?

Don't try and hide things through introspection, it's just security through obscurity.

Public APIs may choose to publish the SDL and disable introspection to protect against introspection attacks.

Document allow list:
No need to disable introspection

Never treat your schema as a secret.
Care MUST be taken.

Handling Malicious Requests

Using **holistic** rule checks

Pagination limits

Should be implemented in resolvers/business-logic.

Even better as a validation rule: abort pre-execution on disallowed limits.

If the limits are hard-coded, no runtime checks are needed.

Query cost + complexity analysis

Attackers can form highly complex requests in small documents.

Attackers can also build very large and time-consuming to execute/validate documents.

Check out IBM's GraphQL Cost Directives Specification
ibm.github.io/graphql-specs/

Factor into rate limiting: allow loads of simple queries, or just a few complex queries - ensure load from a single user is balanced across the needs of all users.

Query cost analysis not needed;
Can act as guardrails.

BEWARE!

RUNTIME ERRORS

Error masking

- Replace all errors with generic errors by default
  - **Don’t** reveal implementation details
  - **Don’t** reveal error codes
  - **Don’t** expose stack traces!
- No need to mask known-safe errors e.g. some data validation errors


FAST AND SECURE
PROTECT YOUR API WITH
TRUSTED
DOCUMENTS
IF YOU CAN, YOU SHOULD
COMPATIBLE WITH
MOST GRAPHQL CLIENTS AND SERVERS


NEW
IMPROVED
TRUST!
ECURE
PROTECT YOUR API WITH
TRUSTED
DOCUMENTS
IF YOU CAN, YOU SHOULD
COMPATIBLE WITH
MOST GRAPHQL CLIENTS AND SERVERS

Trusted document security tips

Careful query design - **only request what you need**.

No unbounded pagination!

Be careful with variables.

```
first: $first
```

use either different docs, or hardcode limits into API or document

```
filter: $filter
```

instead use

```
filter: {
  users: {
    id: {
      greaterThan: $var
} } }
```

Trusted document security tips

Careful query design - **only request what you need**.

No unbounded pagination!

Be careful with variables.

Still validate your documents; e.g. using `gqlcheck`:

- Can add exceptions on a per-document, per-coordinate basis.
- Supports a “baseline” where you can import all existing documents as valid.

Trusted docs, including incorporating into existing projects:
benjie.dev/graphql/trusted-documents

Protecting against malicious queries:
the-guild.dev/graphql/envelop/v3/guides/securing-your-graphql-api
npmjs.com/package/gqlcheck

OWASP security cheatsheet:
cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html

Error masking: the-guild.dev/graphql/yoga-server/docs/features/error-masking

GraphQL bombs:
escape.tech/blog/forging-graphql-bombs-the-2022-version-of-zip-bombs/

And remember: **Trusted Documents: if you can, you should!**

Slides