GraphQL — Cheat Sheet

spec.graphql.org · Relay Cursor Connections · Spring for GraphQL · Strawberry/FastAPI · Apollo/urql · irurueta.com

OPERATION SHAPE + RESPONSE ENVELOPE

```
query GetBook($id: ID!) {
  book(id: $id) { id title author { name } }
}
mutation AddBook($in: BookInput!) {
  addBook(input: $in) { id title }
}
subscription OnBookAdded {
  bookAdded { id title }
}
```

variables sent as separate JSON, not string-interpolated

```
{
  "data": { "book": {...} } | null,
  "errors": [ { "message": "...",
    "locations": [{"line":2,"column":3}],
    "path": ["book","author"],
    "extensions": { "code": "NOT_FOUND" } } ]
}
```

data and **errors** may BOTH be present (partial success). **query** read · **mutation** write · **subscription** long-lived stream, one event per message.

SDL ESSENTIALS

```
type Book implements Node {
  id: ID!
  title: String!
  pages: Int
  rating: Float
  published: Boolean!
  status: Status!
  tags: [String!]!
  author: Author
}
enum Status { DRAFT PUBLISHED ARCHIVED }
interface Node { id: ID! }
union SearchResult = Book | Author
input BookInput { title: String! authorId: ID! }
```

5 scalars: `Int Float String Boolean ID`

`Type` nullable · `Type!` non-null

`[Type]` nullable list · `[Type!]!` non-null list of non-null items

interface/union → polymorphic output; input → object arguments (no methods, no interfaces on fields)

N+1 PROBLEM → DATALOADER

Listing N books, each resolving `author` naively → 1 + N queries.

```
const authorLoader = new DataLoader(async
(ids) =>
  ids.map(id => authorsById.get(id)) // 1
query, all ids
);
author: (book) =>
authorLoader.load(book.authorId)
```

batch: collects `.load()` calls made in the same tick into one call.

cache: dedupes repeated loads within a single request. Create a **new** DataLoader per request — never share/cache across requests.

EXECUTION MODEL

```
type Query { book(id: ID!): Book }
// resolver signature
book: (parent, args, ctx, info) =>
db.find(args.id)
```

parent — previous field's resolved value

args — this field's arguments

ctx — shared per-request state (auth, loaders)

info — schema + query AST metadata

Query: sibling fields resolve in **parallel**.

Mutation: top-level fields resolve **serially**, in document order.

null propagation: an error on a non-null (`!`) field nulls the nearest *nullable* ancestor; the error is recorded in `errors`, not thrown to the client.

SERVING OVER HTTP

```
POST /graphql
Content-Type: application/json
Accept: application/graphql-response+json

{ "query": "...", "variables": {...},
  "operationName": "GetBook" }
```

GET also allowed for cacheable queries (`?query=...&variables=...`).

media types: `application/graphql-response+json` (spec, preferred) · `application/json` (legacy, still common)

status codes: **200** even when `errors` is populated — a well-formed GraphQL response. Use **4xx/5xx** only for request-level failures (malformed JSON, auth, unsupported media type) — never for field-level execution errors.

RELAY CURSOR CONNECTIONS

```
query {
  books(first: 2, after: "Y3Vyc29yOjE=") {
    edges { cursor node { id title } }
    pageInfo {
      hasNextPage hasPreviousPage
      startCursor endCursor
    }
  }
}
```

cursor — opaque, stable pointer to a position in the list (do not parse client-side).

`first/after` → forward paging · `last/before` → backward paging.

`pageInfo.hasNextPage` drives "load more" / infinite scroll.

SECURITY / DEMAND CONTROL

- ☐ **depth limit** — reject deeply nested queries (e.g. max 10)
- ☐ **complexity/cost limit** — weight fields, cap total score per request
- ☐ **timeouts** — enforce a per-request execution deadline
- ☐ **disable introspection** in production (or restrict to trusted callers)
- ☐ **trusted / persisted documents** — client sends a hash id, server resolves it to a pre-approved query; arbitrary query strings rejected
- ☐ **rate-limit by cost**, not just request count
- ☐ **alias/batch limits** — cap aliasing to block amplification attacks

SPRING FOR GRAPHQL

```
@Controller
class BookController {

  @QueryMapping
  Book book(@Argument String id) {...}

  @MutationMapping
  Book addBook(@Argument BookInput in) {...}

  @SchemaMapping(typeName = "Book")
  @BatchMapping
  Map<Book,Author> author(List<Book> books) {
    // one call batches the whole list
  }
}
```

```
spring.graphql.path=/graphql
spring.graphql.graphiql.enabled=true
spring.graphql.cors.allowed-origins=...
```

`@BatchMapping` = built-in DataLoader-backed batching — the N+1 fix above, wired for you.

STRAWBERRY / FASTAPI

```
import strawberry
from strawberry.fastapi import GraphQLRouter

@strawberry.type
class Book:
    id: strawberry.ID
    title: str

@strawberry.type
class Query:
    @strawberry.field
    def book(self, id: strawberry.ID) -> Book:
...

schema = strawberry.Schema(query=Query)

async def context_getter():
    return {"author_loader": AuthorLoader()}

router = GraphQLRouter(schema,
    context_getter=context_getter)
app.include_router(router, prefix="/graphql")
```

`strawberry.dataloader.DataLoader` instance created per request inside `context_getter`, read back via `info.context` in resolvers.

CLIENT ESSENTIALS

Apollo Client (JS)

```
new ApolloClient({
  link: new HttpLink({ uri: "/graphql" }),
  cache: new InMemoryCache()  // normalizes by
id
});
```

`ApolloLink` chain composes middleware: auth → retry → error → http.

urql

```
createClient({ url: "/graphql",
  exchanges: [cacheExchange,
    authExchange(...), fetchExchange] });
```

exchanges are composable middleware; order matters (last exchange must perform the fetch).

JVM — Spring GraphQlClient

```
HttpGraphQlClient client =
  HttpGraphQlClient.create(webClient);
client.document(query)
  .retrieve("book").toEntity(Book.class);
```