GraphQL

Ask for precisely shaped data.

01 Select an object

```
{
  user(id: "1") {
    id name
  }
}
```

Operation: objects need subfields; scalars do not. This anonymous query uses the User API in sections 10-14. Response keys follow the selection.

02 Name an operation

```
query People {
  users(limit: 2) {
    id
    name
  }
}
```

Operation: People names the query. Choose one operation if a document has several. limit belongs to this example API; pagination is not automatic.

03 Arguments & aliases

```
query Compare {
  first: user(id: "1") { name }
  second: user(id: "2") {
    name
  }
}
```

Operation: arguments customize a field. Aliases rename response keys and permit different arguments. first and second each contain name.

04 Declare variables

```
query Person($id: ID!) {
  user(id: $id) {
    id name
  }
}
```

Operation: $ declares/uses a variable. Its type must fit the argument; ID! requires non-null. Send values separately, without interpolating query text.

05 Send variables (JSON)

```
{
  "operationName": "Person",
  "variables": {
    "id": "1"
  }
}
```

JSON request fields alongside section 04's query text. Variable keys omit $. Values use JSON syntax. operationName selects an operation, not a schema field.

06 Reuse fragments

```
fragment Card on User {
  id name
}
query Profile {
  user(id: "1") { ...Card }
}
```

Operation document: ...Card spreads a fragment on a compatible type. Send its definition with the operation. Fragments reuse selections, not schema definitions.

07 Conditional fields

```
query Details($show: Boolean! = false) {
  user(id: "1") {
    name
    email @include(if: $show)
  }
}
```

Operation: @include selects when true; @skip omits when true. Skipped fields are absent. Omitting $show uses false. These are not authorization checks.

08 Nested objects & lists

```
query Friends {
  user(id: "1") {
    name
    friends { id name }
  }
}
```

Operation: selections follow schema relationships. friends returns a JSON array. Fewer fields shape the response, without guaranteeing database cost.

09 Multiple root fields

```
query Overview {
  user(id: "1") { name }
  users(limit: 2) {
    id name
  }
}
```

Operation: select several root fields in one query. They may run in parallel; do not rely on execution order. This response has user and users keys.

GraphQL

Define types and handle results.

10 Object types (SDL)

```
type User {
  id: ID!
  name: String!
  email: String
  role: Role!
  friends: [User!]!
}
```

Schema definition language (SDL): ! means non-null. email may be null; Role is in section 13.

11 Query root & lists (SDL)

```
type Query {
  user(id: ID!): User
  users(
    limit: Int = 10
  ): [User!]!
}
```

SDL: Query exposes query roots. [User!]! allows [], but no null list or items. [User] permits both. Argument omission uses its declared default.

12 Built-in scalar types

Int	Signed 32-bit integer
Float	Double-precision number
String	Text
Boolean	true or false
ID	Identifier serialized as a string

ID accepts string or integer input. Custom scalars need server-defined validation and serialization.

13 Enums & inputs (SDL)

```
enum Role { USER ADMIN }
input UserInput {
  name: String!
  email: String
  role: Role = USER
}
```

SDL: enums constrain values; inputs group arguments, distinct from output objects. Inline enum values use ADMIN; JSON variable values use "ADMIN".

14 Mutation root (SDL)

```
type Mutation {
  createUser(
    input: UserInput!
  ): User!
}
```

SDL: mutation root fields run serially, without an automatic transaction. The server defines write behavior, validation, and authorization.

15 Create data

```
mutation Add($input: UserInput!) {
  createUser(input: $input) {
    id
    name
    role
  }
}
```

Operation: variables can be {"input":{"name":"Maya"}}. Select return fields as in a query. Data changes only when the server executes this operation.

16 Response data (JSON)

```
{
  "data": {
    "user": {"id": "1", "name": "Maya"}
  }
}
```

Response to section 01: data mirrors selections and aliases. IDs serialize as strings. A nullable field can return null without an error.

17 Errors & partial data

```
{
  "data": {"user": null},
  "errors": [{
    "message": "User lookup failed",
    "path": ["user"]
  }]
}
```

JSON: errors can accompany partial data. Non-null failures propagate to a nullable parent. Parse/validation failures omit data. Check errors, not only HTTP status.

18 A JSON request envelope

```
{
  "query": "query Kind { __typename }",
  "operationName": "Kind",
  "variables": {}
}
```

Typical POST JSON body; __typename returns the root type name. Use the API's documented endpoint, content type, and headers. Authentication and pagination are API conventions.