GraphQL

Ask for specific data like Id,name and Author from Courses–

Query {

Courses {

Id,

Name,

Author

}

}

For specific course –

Query {

Courses (id = “1”) {

Id,

Name,

Author

}

}

Fetch nested data

Query {

Courses (id = “1”) {

Id,

Name,

Author {

Name,

Id,

Courses{

Id,

Title,

}

}

}

}

Mutations

**🔁 What is a Mutation in GraphQL?**

In GraphQL, a **mutation** is used to **modify data** on the server (like create, update, or delete records). It’s like a **POST, PUT, or DELETE** in REST APIs.

Whereas **queries** fetch data (read-only), **mutations** change it.

**🛠 Example of a Mutation**

Let’s say we have a system for managing games and reviews.

**📌 Mutation to Add a Review:**

mutation AddReview($gameId: ID!, $rating: Int!, $content: String!) {

addReview(gameId: $gameId, rating: $rating, content: $content) {

id

rating

content

}

}

**📦 Variables:**

{

"gameId": "2",

"rating": 5,

"content": "Amazing gameplay!"

}

This mutation does 3 things:

* Calls the addReview mutation on the server.
* Passes the required inputs: gameId, rating, content.
* Returns the new review's id, rating, and content.

**📚 Where Do Mutations Live?**

In your GraphQL schema:

type Mutation {

addReview(gameId: ID!, rating: Int!, content: String!): Review

deleteReview(id: ID!): Boolean

}

Just like Query is for reading data, Mutation is for modifying it.

**✅ Tips to Remember**

| **Queries** | **Mutations** |
| --- | --- |
| Read-only | Modify data |
| Can be cached | Usually not cached |
| Use query | Use mutation keyword |

Want help writing a mutation for your specific schema? Drop the schema or tell me what you're trying to do (like "create a user" or "delete a review"), and I’ll walk you through it.

**Setup**[**#**](https://docs.nestjs.com/first-steps#setup)

Setting up a new project is quite simple with the [**Nest CLI**](https://docs.nestjs.com/cli/overview). With **[npm](https://www.npmjs.com/" \t "_blank)** installed, you can create a new Nest project with the following commands in your OS terminal:

Install the CLI globally using the npm install -g command

$ npm install -g @nestjs/cli

To create, build and run a new basic Nest project in development mode, go to the folder that should be the parent of your new project, and run the following commands:

$ nest new my-nest-project

$ cd my-nest-project

$ npm run start:dev

In your browser, open [**http://localhost:3000**](http://localhost:3000/) to see the new application running. The app will automatically recompile and reload when you change any of the source files.

Here's a brief overview of those core files:

app.controller.ts A basic controller with a single route.

app.controller.spec.ts The unit tests for the controller.

app.module.ts The root module of the application.

app.service.ts A basic service with a single method.

main.ts The entry file of the application which uses the core function NestFactory to create a Nest application instance.

What is controller?

Controllers are defined using classes and decorators. The @Controller() decorator is used to designate a class as a controller and to specify an optional route path prefix. Within the class, methods are defined to handle specific HTTP requests, using decorators like @Get(), @Post(), @Put(), and @Delete().​[NestJS Docs+1NestJS Docs+1](https://docs.nestjs.com/openapi/operations?utm_source=chatgpt.com" \t "_blank)

**What is CatsController?**

It’s a **made-up controller** that handles requests related to "cats"—like getting a list of cats or finding one cat by ID.

Think of it as a part of an app that could manage a **cat database**. So the controller responds to API calls like:

* GET /cats – get all cats
* GET /cats/1 – get the cat with ID 1

import { Controller, Get } from '@nestjs/common';

@Controller('cats')

export class CatsController {

@Get()

findAll(): string {

return 'This action returns all cats';

}

}

**What it means:**

* @Controller('cats'): This sets the base route to /cats
* @Get(): This handles a GET request to /cats
* findAll() method: It runs when someone sends a GET request to /cats
* So if you open a browser and go to http://localhost:3000/cats, it would show:

This action returns all cats

The name CatsController is just a sample. In a real app, you might have:

* UsersController for managing users
* ProductsController for products
* OrdersController for orders
* **Route Handlers and Parameters**
* NestJS allows you to define route handlers that can accept parameters. For instance, you can capture route parameters using the @Param() decorator.​

import { Controller, Get, Param } from '@nestjs/common';

@Controller('cats')

export class CatsController {

@Get(':id')

findOne(@Param('id') id: string): string {

return `This action returns a cat with id ${id}`;

}

}

**xplanation:**

**🔹 import { Controller, Get, Param } from '@nestjs/common';**

* This pulls in 3 things from NestJS:
  + Controller: To define a controller (a class that handles incoming requests).
  + Get: A decorator to handle GET requests (like visiting a page).
  + Param: Used to get data from the **URL**, like the cat's ID.

**🔹 @Controller('cats')**

* This tells NestJS: "This controller handles routes that start with /cats".
* So this class responds to URLs like:
  + /cats/1
  + /cats/abc

**🔹 export class CatsController { ... }**

* This defines a controller class called CatsController.
* Inside it, we’ll put the logic for handling requests.

**🔹 @Get(':id')**

* This means: "When someone sends a GET request to /cats/:id"
  + :id is a **route parameter** (a placeholder for any value).

Example:

* /cats/5 – here id = 5
* /cats/banana – here id = banana

**🔹 findOne(@Param('id') id: string): string**

* findOne is the function that runs when the route /cats/:id is hit.
* @Param('id') pulls the id value from the URL.
* id: string means the id will be treated as a string.
* The function just **returns a message** that includes the cat's ID.

**💡 Example in Action:**

If you visit:

bash

CopyEdit

GET /cats/123

You’ll get:

bash

CopyEdit

This action returns a cat with id 123

Here, a GET request to /cats/1 would invoke the findOne method with id set to '1'.​

**Testing Controllers**

NestJS provides utilities to test controllers effectively. You can use the @nestjs/testing package to create a testing module and write unit tests for your controllers.

**Updated Code with POST:**

import { Controller, Get, Post, Param, Body } from '@nestjs/common';

@Controller('cats')

export class CatsController {

@Get(':id')

findOne(@Param('id') id: string): string {

return `This action returns a cat with id ${id}`;

}

@Post()

create(@Body() createCatDto: any): string {

return `This action adds a new cat with name ${createCatDto.name}`;

}

}

**What’s New:**

**🔹 @Post()**

* Handles a **POST** request to /cats.
* This is where we receive new cat data.

**🔹 @Body() createCatDto: any**

* @Body() gets the data sent in the body of the POST request.
* createCatDto is just a variable to hold that data (DTO = Data Transfer Object).
* We're using any to keep it simple for now.

**Example POST Request:**

* Send a POST request to /cats with this JSON body:

{

"name": "Whiskers"

}

The response will be:

This action adds a new cat with name Whiskers

add validation,

NestJS uses a powerful package called **class-validator** with **DTOs (Data Transfer Objects)** to validate incoming data.

Install required packages (if you haven't already)

npm install class-validator class-transformer

Create a DTO file: create-cat.dto.ts

import { IsString, IsNotEmpty } from 'class-validator';

export class CreateCatDto {

@IsString()

@IsNotEmpty()

name: string;

}

This says:

* name must be a string
* name cannot be empty

Update the Controller:

import { Controller, Get, Post, Param, Body } from '@nestjs/common';

import { CreateCatDto } from './create-cat.dto'; // <-- import your DTO

import { ValidationPipe, UsePipes } from '@nestjs/common';

@Controller('cats')

export class CatsController {

@Get(':id')

findOne(@Param('id') id: string): string {

return `This action returns a cat with id ${id}`;

}

@Post()

@UsePipes(new ValidationPipe()) // <-- apply validation

create(@Body() createCatDto: CreateCatDto): string {

return `This action adds a new cat with name ${createCatDto.name}`;

}

}

**Test It:**

Send a POST request to /cats:

{ "name": "" }

You’ll get an error like:

{

"statusCode": 400,

"message": ["name should not be empty"],

"error": "Bad Request"

}

Why we use Validation Pipe -

import { ValidationPipe, UsePipes } from '@nestjs/common';

**ValidationPipe**

* A **built-in NestJS class** that automatically **checks if incoming data is valid**, based on the rules defined in a **DTO** (like CreateCatDto).
* It uses class-validator under the hood.

Think of it like a **security guard** for your data:

* If the data is valid ✅ → it lets it through.
* If not ❌ → it blocks it and returns an error.

**UsePipes()**

* A **decorator** that lets you **attach a pipe** (like ValidationPipe) to a route handler.
* It tells NestJS to apply that pipe **before** running the function (in this case, before creating a cat).

@Post()

@UsePipes(new ValidationPipe())

create(@Body() createCatDto: CreateCatDto): string {

return `This action adds a new cat with name ${createCatDto.name}`;

}

Here’s what happens:

1. A POST request hits the /cats endpoint.
2. @UsePipes(new ValidationPipe()) kicks in.
3. It checks if the data in createCatDto matches the rules.
4. If it passes → create() runs.
5. If it fails → an error is returned.

**What is a Provider?**

A **provider** is any class that **can be injected and used elsewhere** in your application using **dependency injection (DI)**. Providers are a core concept in Nest. Many of the basic Nest classes, such as services, repositories, factories, and helpers, can be treated as providers. The key idea behind a provider is that it can be **injected** as a dependency, allowing objects to form various relationships with each other. The responsibility of "wiring up" these objects is largely handled by the Nest runtime system.

Providers are used to:

* Share **business logic** (like services)
* Access databases
* Perform tasks like sending emails, making API calls, etc.

In short:  
📦 A **provider is a class that provides a value or functionality** to be used in other parts of your app.

**Common Provider Example: A Service**

import { Injectable } from '@nestjs/common';

@Injectable()

export class CatsService {

private cats = [];

findAll() {

return this.cats;

}

create(cat: any) {

this.cats.push(cat);

}

}

* @Injectable() tells NestJS this class **can be injected** as a dependency.
* It holds logic for finding and creating cats.

Using the Provider in a Controller

import { Controller, Get, Post, Body } from '@nestjs/common';

import { CatsService } from './cats.service';

@Controller('cats')

export class CatsController {

constructor(private readonly catsService: CatsService) {}

@Get()

findAll() {

return this.catsService.findAll();

}

@Post()

create(@Body() cat: any) {

this.catsService.create(cat);

return 'Cat added!';

}

}

 The CatsController **injects** CatsService using the constructor.

 NestJS automatically provides an instance of CatsService.

**Recap:**

| **Term** | **What it is** |
| --- | --- |
| @Injectable | Marks a class as a provider (DI ready) |
| Provider | A class that can be injected and reused |
| Service | A type of provider that holds business logic |

**Modules**

A module is a class that is annotated with the @Module() decorator. This decorator provides metadata that **Nest** uses to organize and manage the application structure efficiently.

Quick start

Resolvers

**Resolvers**

Resolvers provide the instructions for turning a **[GraphQL](https://graphql.org/" \t "_blank)** operation (a query, mutation, or subscription) into data. They return the same shape of data we specify in our schema -- either synchronously or as a promise that resolves to a result of that shape. Typically, you create a **resolver map** manually. The @nestjs/graphql package, on the other hand, generates a resolver map automatically using the metadata provided by decorators you use to annotate classes. To demonstrate the process of using the package features to create a GraphQL API, we'll create a simple authors API

Mutations

In **GraphQL**, a **mutation** is used to **change data** on the server — like **creating**, **updating**, or **deleting** something.

Think of it like:

* **Query** = "I want to *get* data"
* **Mutation** = "I want to *change* data"

Example: A Mutation to Add a Cat

mutation {

createCat(name: "Whiskers", age: 2) {

id

name

}

}

This sends a request to **create a new cat** with name "Whiskers" and age 2.

Server-side: How it might be defined

@Resolver()

export class CatsResolver {

@Mutation(() => Cat)

createCat(@Args('name') name: string, @Args('age') age: number): Cat {

return this.catService.create({ name, age });

}

}

 @Mutation() marks this function as a mutation.

 It takes name and age as input.

 Returns the newly created Cat.

**Real-world analogy:**

| **Action** | **Query or Mutation?** |
| --- | --- |
| Get user info | Query |
| Add a new user | Mutation |
| Delete a product | Mutation |
| Update an email | Mutation |
| List all comments | Query |

**What is a Subscription in GraphQL?**

A **subscription** in GraphQL is used for **real-time communication**.  
It lets the server **push updates to the client** when something changes — instead of the client constantly asking (polling) for new data.

**🔁 Comparison:**

| **Type** | **Purpose** | **Example** |
| --- | --- | --- |
| Query | Read data | Get a list of cats |
| Mutation | Write/change data | Add a new cat |
| Subscription | Real-time updates | Get notified when a new cat is added |

**🧠 Real-world analogy:**

Like YouTube Live Chat: you **subscribe** to messages, and new ones just pop in automatically — you don’t have to refresh. 🎥💬

**📦 GraphQL Subscription Example**

Client sends:

graphql

CopyEdit

subscription {

catAdded {

name

age

}

}

Whenever a new cat is added (on the server), the client **automatically receives** that new data.

**🔧 Tech Behind the Scenes:**

* Subscriptions usually use **WebSockets**, not HTTP.
* On the server side, you use tools like @nestjs/graphql and @nestjs/apollo with WebSocket support.

**What is a Resolver?**

A **resolver** is a function that **answers a GraphQL query, mutation, or subscription**.

In NestJS, **resolver classes** handle how you **fetch or modify data** based on what the client is asking.

You can think of resolvers as:

🧭 The **"brain" behind each GraphQL operation** — they tell the server *what to do* when someone runs a query or mutation.

**Example:**

Let’s say the client sends this GraphQL query

query {

getAllCats {

name

age

}

}

NestJS looks at your **resolver class** to figure out what getAllCats means and how to respond.

Resolver in NestJS

import { Resolver, Query } from '@nestjs/graphql';

import { Cat } from './entities/cat.entity';

import { CatsService } from './cats.service';

@Resolver(() => Cat)

export class CatsResolver {

constructor(private catsService: CatsService) {}

@Query(() => [Cat])

getAllCats(): Cat[] {

return this.catsService.findAll();

}

}

| **Part** | **Meaning** |
| --- | --- |
| @Resolver(() => Cat) | Tells Nest this resolver is for the Cat type |
| @Query(() => [Cat]) | Marks the function as a GraphQL query that returns a list of cats |
| getAllCats() | This is the actual resolver function |
| catsService.findAll() | Fetches the data from a service (or database) |

There are 3 main types of resolvers:

| **Decorator** | **What it does** | **Example** |
| --- | --- | --- |
| @Query() | Read data | getAllCats, getCatById |
| @Mutation() | Change data | createCat, deleteCat |
| @Subscription() | Real-time updates | catAdded, catDeleted |

**Recap:**

* **Resolvers** are the heart of a GraphQL API.
* They **connect GraphQL operations (query/mutation/subscription)** to your actual logic (services, database, etc).
* In NestJS, you create them as **classes with decorators** like @Query(), @Mutation(), etc.

Unions and Enums

**🔶 ENUMS**

**🧠 What is an Enum?**

An **enum** (short for “enumeration”) is a list of **named constant values**.

It’s like saying:

“This field can only be one of a few specific options.”

**📦 Example: Cat Breed Enum**

// cats/enums/breed.enum.ts

import { registerEnumType } from '@nestjs/graphql';

export enum Breed {

PERSIAN = 'PERSIAN',

SIAMESE = 'SIAMESE',

MAINE\_COON = 'MAINE\_COON',

}

registerEnumType(Breed, {

name: 'Breed', // this name will show in GraphQL schema

});

Then in a DTO or entity:

@ObjectType()

export class Cat {

@Field()

name: string;

@Field(() => Breed)

breed: Breed;

}

Now in GraphQL Playground, you’ll see:

enum Breed {

PERSIAN

SIAMESE

MAINE\_COON

}

**🔁 UNIONS**

**🧠 What is a Union?**

A **union** lets you return **multiple different types** from a single field.

It's like saying: “This result might be a Cat, or it might be a Dog.”

Useful when a query can return **different kinds of objects**.

**📦 Example: Union of Cat and Dog**

// animals/animal.union.ts

import { createUnionType } from '@nestjs/graphql';

import { Cat } from './entities/cat.entity';

import { Dog } from './entities/dog.entity';

export const AnimalResult = createUnionType({

name: 'AnimalResult',

types: () => [Cat, Dog],

resolveType(value) {

if ('clawSharpness' in value) {

return Cat;

}

if ('barkVolume' in value) {

return Dog;

}

return null;

},

});

Then use it in a resolver:

@Query(() => AnimalResult)

getRandomAnimal(): typeof AnimalResult {

return Math.random() > 0.5

? { name: 'Whiskers', clawSharpness: 10 }

: { name: 'Rex', barkVolume: 5 };

}

**🧩 Summary**

| **Feature** | **Purpose** | **Example** |
| --- | --- | --- |
| **Enum** | Fixed set of options | Breed.PERSIAN, Breed.SIAMESE |
| **Union** | Return multiple types from one field | Might return a Cat or a Dog |

Let me know if you want to create a working union or enum in your own project! I can guide you line by line. 🧑‍💻

<https://the-guild.dev/graphql/apollo-angular/docs>

Get started

**nstallation with Angular Schematics**

The simplest way to get started with Apollo Angular is by executing the schematic like this:

ng add apollo-angular

Queries <https://the-guild.dev/graphql/apollo-angular/docs/data/queries>

import { Apollo, gql } from 'apollo-angular';

import { Subscription } from 'rxjs';

import { Component, OnDestroy, OnInit } from '@angular/core';

// We use the gql tag to parse our query string into a query document

const GET\_POSTS = gql`

query GetPosts {

posts {

id

title

}

}

`;

@Component({

// ...

})

class PostsComponent implements OnInit, OnDestroy {

loading: boolean;

posts: any;

private querySubscription: Subscription;

constructor(private readonly apollo: Apollo) {}

ngOnInit() {

this.querySubscription = this.apollo

.watchQuery<any>({

query: GET\_POSTS,

})

.valueChanges.subscribe(({ data, loading }) => {

this.loading = loading;

this.posts = data.posts;

});

}

ngOnDestroy() {

this.querySubscription.unsubscribe();

}

}

You're doing great — this is a solid example of how Angular integrates with GraphQL using apollo-angular. Let’s break it down step by step in a beginner-friendly way.

**🔍 What this code does:**

It defines a **component** that **fetches a list of posts** using a GraphQL query and displays them in an Angular app. It uses **Apollo Client** to handle the GraphQL part.

**🧩 Section-by-Section Explanation:**

**1. Imports**

import { Apollo, gql } from 'apollo-angular';

import { Subscription } from 'rxjs';

import { Component, OnDestroy, OnInit } from '@angular/core';

* Apollo: A service from Apollo Angular to talk to your GraphQL server.
* gql: A tag function to define a GraphQL query.
* Subscription: From RxJS, used to manage and clean up data streams.
* OnInit and OnDestroy: Angular lifecycle interfaces (what to do when the component starts and ends).

**2. GraphQL Query**

const GET\_POSTS = gql`

query GetPosts {

posts {

id

title

}

}

`;

This defines a GraphQL **query** to fetch a list of posts with their id and title.  
GraphQL is very "ask for what you need", so you specify the fields you want.

**3. Component Declaration**

@Component({

// ...

})

class PostsComponent implements OnInit, OnDestroy {

This is a basic Angular component named PostsComponent. It uses:

* OnInit: to run code when the component loads.
* OnDestroy: to clean up when the component is removed.

**4. Class Properties**

loading: boolean;

posts: any;

private querySubscription: Subscription;

* loading: shows whether data is being loaded.
* posts: stores the list of posts fetched from the server.
* querySubscription: used to keep track of the data stream so we can unsubscribe later.

**5. Constructor**

constructor(private readonly apollo: Apollo) {}

Injects the Apollo service into the component so we can use it to run the query.

**6. ngOnInit (Component Initialization)**

ngOnInit() {

this.querySubscription = this.apollo

.watchQuery<any>({

query: GET\_POSTS,

})

.valueChanges.subscribe(({ data, loading }) => {

this.loading = loading;

this.posts = data.posts;

});

}

Here’s what happens:

* watchQuery runs the GET\_POSTS query and watches for updates.
* .valueChanges is an **Observable** stream of the query results.
* .subscribe(...) listens for new data.
* When data arrives, it:
  + Updates this.loading with the current loading state
  + Saves the posts into the component’s posts variable

**7. ngOnDestroy (Cleanup)**

ngOnDestroy() {

this.querySubscription.unsubscribe();

}

When the component is removed from the screen, this stops the subscription to avoid memory leaks.

Extra explanation

Absolutely! Let's break this ngOnInit() method down **line by line**, beginner-style 👇

**🧩 Full Code We're Explaining:**

ngOnInit() {

this.querySubscription = this.apollo

.watchQuery<any>({

query: GET\_POSTS,

})

.valueChanges.subscribe(({ data, loading }) => {

this.loading = loading;

this.posts = data.posts;

});

}

**🧠 What is ngOnInit()?**

* This is an Angular **lifecycle method**.
* It runs **automatically when the component is created (initialized)**.
* It's a good place to start loading data from a server — like we're doing here!

**🔍 Line-by-Line Explanation**

**✅ this.querySubscription = this.apollo**

* You're calling a service called **Apollo**, which is used to connect to a **GraphQL API**.
* This is the service you injected in the constructor earlier.

**✅ .watchQuery<any>({ query: GET\_POSTS })**

* watchQuery sends a **GraphQL query** to your server and **"watches" it for changes**.
* It takes an object: { query: GET\_POSTS } — this is your GraphQL query to fetch posts.
* The <any> here is a quick way to say “don’t worry about strict typing” for now. Later you can replace it with a real type.

**✅ .valueChanges**

* This returns an **Observable** — like a stream of data.
* Whenever new data comes in (e.g., from server or cache), this stream "emits" the new values.

**✅ .subscribe(({ data, loading }) => { ... })**

* You're subscribing to the stream — which means:
  + “Whenever there’s new data, run this function.”
* { data, loading } is an object coming from Apollo, where:
  + data: the actual result from the query
  + loading: a boolean that tells you if the request is still in progress

**✅ Inside the subscribe() function:**

this.loading = loading;

this.posts = data.posts;

* You update your component’s properties:
  + this.loading → shows whether data is still loading
  + this.posts → stores the actual list of posts from the server

**✅ What is this whole thing doing?**

When the component loads:

1. It sends a request to get posts using GET\_POSTS.
2. It **listens** for results with subscribe.
3. Once data arrives:
   * It updates loading (true → false)
   * It fills in the posts list

Let me know if you want to add a loading spinner or show the posts in your template next — happy to help!

**🧠 Summary**

This component:

* Runs a GraphQL query on load
* Watches for data updates
* Stores the results
* Cleans up when the component is destroyed

Would you like me to show how you could display the posts in the HTML part of this component too?

Great follow-up! You're getting into the deeper — but really useful — parts of Apollo and GraphQL in Angular.

**🔍 What is QueryRef?**

In Apollo Angular, **QueryRef** stands for **"Query Reference"**.

It’s an **object returned by watchQuery()**, and it gives you control over a running query.

**✅ Why use QueryRef?**

When you call watchQuery(), you can either:

* **Subscribe directly** (like you did earlier)
* Or **save the query as a QueryRef** so you can:
  + **Refetch data**
  + **Fetch more items (pagination)**
  + **Stop or restart the query**
  + **Access the current state anytime**

**🧠 Think of QueryRef like this:**

It's like a **remote control** for your GraphQL query — you can pause, play, update, or re-run it as needed.

**🧪 Example Use:**

import { Apollo, QueryRef } from 'apollo-angular';

queryRef: QueryRef<any>;

posts: any[];

ngOnInit() {

this.queryRef = this.apollo.watchQuery({

query: GET\_POSTS

});

this.queryRef.valueChanges.subscribe(({ data }) => {

this.posts = data.posts;

});

}

Now that you have this.queryRef, you can do things like:

**🔁 Refetch the data manually:**

this.queryRef.refetch();

**➕ Fetch more items (pagination example):**

this.queryRef.fetchMore({

variables: { offset: 10 },

updateQuery: (prev, { fetchMoreResult }) => {

return {

posts: [...prev.posts, ...fetchMoreResult.posts]

};

}

});

**❌ Stop or unsubscribe:**

You can unsubscribe from valueChanges using the usual .unsubscribe().

**🔄 Use Case Recap**

| **Action** | **Method** |
| --- | --- |
| Get current value | queryRef.valueChanges |
| Refetch manually | queryRef.refetch() |
| Fetch more (e.g., load more posts) | queryRef.fetchMore() |
| Update variables or polling | queryRef.setOptions() |

Would you like an example using refetch() to reload posts when a user clicks a button?

Mutations <https://the-guild.dev/graphql/apollo-angular/docs/data/mutations>

**Mutations**

In addition to fetching data using queries, Apollo also handles GraphQL mutations. Mutations are identical to queries in syntax, the only difference being that you use the keyword mutation instead of query to indicate that the operation is used to change the dataset behind the schema.

mutation upvotePost { upvotePost(postId: 12) { id votes }}

GraphQL’s mutations consist of two parts:

1. The mutation name with arguments (upvotePost), which represents the actual operation to be done on the server
2. The fields you want back from the result of the mutation to update the client (id and vote)

The result of the above mutation might be:

{ "data": { "upvotePost": { "id": 12, "votes": 123 } }}

When we use mutations in Apollo, the result is typically integrated into the cache automatically [based on the id of the result](https://the-guild.dev/graphql/apollo-angular/docs/caching/interaction#normalization-with-dataidfromobject), which in turn updates UI automatically, so we don’t explicitly handle the results ourselves. In order for the client to correctly do this, we need to ensure we select the correct fields (as in all the fields that we care about that may have changed).

**Basic Mutations**

Using Apollo it’s easy to call mutation. You can simply use mutate method.

import { Apollo, gql } from 'apollo-angular';

import { Component } from '@angular/core';

const UPVOTE\_POST = gql`

mutation UpvotePost {

upvotePost(postId: 12) {

id

votes

}

}

`;

@Component({

// ...

})

class UpvotePostComponent {

constructor(private readonly apollo: Apollo) {}

newRepository() {

this.apollo.mutate({ mutation: UPVOTE\_POST }).subscribe();

}

}

**🧱 This code does the following:**

* Sends a **GraphQL mutation** to upvote a post with a specific postId.
* Uses Apollo’s mutate() method to perform that mutation from the Angular component.

**🧩 Code Breakdown:**

**✅ Imports**

import { Apollo, gql } from 'apollo-angular';

import { Component } from '@angular/core';

* Apollo: The main service for interacting with GraphQL.
* gql: A tag used to write GraphQL queries or mutations in JavaScript/TypeScript.
* Component: From Angular — to define your Angular component.

**✅ GraphQL Mutation**

const UPVOTE\_POST = gql`

mutation UpvotePost {

upvotePost(postId: 12) {

id

votes

}

}

`;

This defines a GraphQL **mutation** named UpvotePost.

What it does:

* Calls the GraphQL mutation called upvotePost
* Sends a postId: 12 to the server (i.e., "upvote post #12")
* Asks for a response with id and votes — the updated post ID and new vote count

🔁 **Mutation ≠ Query**

* Queries = get data
* Mutations = change data (e.g. add, update, delete)

**✅ Angular Component**

@Component({

// ...

})

class UpvotePostComponent {

This defines the Angular component where the mutation will be triggered.

**✅ Constructor with Apollo**

constructor(private readonly apollo: Apollo) {}

Injects the Apollo service so we can call mutate() from it.

**✅ Method to Run the Mutation**

newRepository() {

this.apollo.mutate({ mutation: UPVOTE\_POST }).subscribe();

}

Here’s what this method does:

1. Calls apollo.mutate() and passes the UPVOTE\_POST mutation
2. subscribe() is used to **actually execute the mutation**
   * Without it, nothing would happen (like turning on a faucet with no cup under it!)

**Note**: This code doesn’t do anything with the result — it just fires the mutation. You can also capture the response like this:

this.apollo.mutate({ mutation: UPVOTE\_POST }).subscribe(result => {

console.log(result.data); // You can do something with the response

});

**✅ What you might improve**

* **Make postId dynamic** instead of hardcoding 12
* **Handle success/error** to show messages to users
* **Update the UI** after mutation (e.g., show new vote count)

**🧠 Summary**

| **Feature** | **What It Does** |
| --- | --- |
| gql | Defines the GraphQL mutation |
| apollo.mutate() | Sends the mutation request |
| subscribe() | Executes the mutation and listens for the result |
| Mutation itself | Tells the server: “Hey, upvote post #12!” |

**🔍 What Each Module Does**

**✅ NgModule from @angular/core**

* **Defines Angular modules**, which are the building blocks of an Angular app.
* Used to group components, services, and other modules into a cohesive unit.

**✅ BrowserModule from @angular/platform-browser**

* Required in **every browser-based Angular app**.
* Provides essential services for running and rendering your app in a browser.

**✅ HttpClientModule from @angular/common/http**

* Enables Angular’s **HTTP client**, which lets your app make HTTP requests (like GET, POST).
* Necessary for Apollo to make network calls under the hood.

**✅ ApolloModule from apollo-angular**

* Provides the **Apollo service**, which is the core of interacting with your GraphQL server.
* You use it to set up the Apollo client in your app and inject it into services/components.

**✅ HttpLinkModule from apollo-angular-link-http**

* Provides HttpLink, a service to connect Apollo Client with Angular's HttpClient.
* It’s how Apollo actually sends your GraphQL queries/mutations over HTTP.

**✅ AppComponent from './app.component'**

* This is your **root component**, usually contains the app’s main layout.
* It’s declared in your AppModule and bootstrapped to launch the app.

**🧠 Summary**

| **Module** | **Purpose** |
| --- | --- |
| NgModule | Define a module in Angular |
| BrowserModule | Run the app in a web browser |
| HttpClientModule | Enable HTTP requests |
| ApolloModule | Core GraphQL functionality |
| HttpLinkModule | Bridge between Apollo and Angular HTTP client |
| AppComponent | Root component of your app |

Let me know if you want a complete example of setting up Apollo in AppModule (including ApolloClientOptions).