Integrating TypeScript with GraphQL

November 11, 2021 · 11 min read

***Editor's note**: This post was last updated on 29 November 2021.*

TypeScript is a typed superset of JavaScript that was designed to solve many of the pain points of writing applications in JavaScript. TypeScript compiles down to plain JavaScript and is backwards-compatible with a few edge cases, making it a great target for older environments.

With features like interfaces, generics, optional type-checking, and type inference, as well as the latest features from ES6 like decorators, async/await, and more, TypeScript stands out greatly when it comes to developer productivity.

On the other hand, GraphQL is a query language for APIs that uses its type system to describe data fields, preventing over and under-fetching. Additionally, GraphQL can greatly help in the area of API versioning.

In this tutorial, we'll build an API to integrate TypeScript with GraphQL using the TypeGraphQL library, which simplifies creating GraphQL APIs in Node.js. TypeGraphQL automatically creates GraphQL schema definitions from TypeScript classes with decorators, which were introduced along with reflection to avoid the need for schema definition files.

Although we could write custom decorators to suit our specific needs based on a project's requirements, for this tutorial, the decorators provided for TypeGraphQL will suffice.

Prerequisites

To follow along with this tutorial, you'll need:

- Familiarity with TypeScript and GraphQL
- Node.js installed on your machine
- TypeScript installed globally on your machine

To install TypeScript, you can run the following command using either npm or Yarn on your terminal or your command prompt:

```
npm install -g typescript
```

The command above installs the latest TypeScript compiler, `tsc` , on our system path, which will be useful when we compile and run our code. `tsc` takes a TypeScript file ending with the `.ts` extension and returns an equivalent JavaScript file with the `.js` extension. We'll learn more about this command later on.

You can access the full code for this tutorial on the GitHub repo.

Table of contents

Let's get started!

Bootstrapping our TypeScript GraphQL application

First, let's create a new directory and name it. We can then initialize a new `package.json` file with the `npm init` command. When we're done installing our project's dependencies, our `package.json` file should look like the code block below:

```
  "devDependencies": {
    "@types/express": "^4.17.3",
    "@types/graphql": "^14.5.0",
    "@types/node": "^13.9.0",
    "nodemon": "^2.0.2",
    "ts-node": "^8.6.2",
    "typescript": "^3.8.3"
  },
  "dependencies": {
    "@typegoose/typegoose": "^6.4.0",

    "apollo-server-express": "^2.11.0",
    "class-validator": "^0.11.0",
    "express": "^4.17.1",
    "graphql": "^14.6.0",
    "reflect-metadata": "^0.1.13",
    "type-graphql": "^0.17.6"
  }
}
```

Installing dependencies

Now, let's install our project's required dependencies. Run the following command in your terminal:

```
npm install type-graphql reflect-metadata graphql express class-validator
apollo-server-express apollo-server-core mongoose @typegoose/typegoose --save
```

As previously mentioned, we'll use the `type-graphql` framework to build our API with TypeScript and GraphQL. TypeGraphQL includes advanced features like automatic validation, dependency injection, authorization, and inheritance, and it allows us to define our GraphQL schema types and interfaces using TypeScript classes and decorators.

The `reflect-metadata` package adds a polyfill for the experimental Metadata API support for TypeScript. Currently, TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators, meaning we'll need to enable support for this library in our `tsConfig.json` file.

`apollo-server-express` is the Express and Connect integration of a GraphQL server, which we'll use to bootstrap a simple GraphQL server with Express.

We'll use the `class-validator` library, which allows the use of decorator and non decorator-based validation with TypeGraphQL, to validate our schema fields. The `mongoose` package is the MongoDB object data mapper (ODM), while `@typegoose/typegoose` allows us to define Mongoose models using TypeScript classes.

Now, let's install the following TypeScript types as dev dependencies, `types/express` , and `@types/node` . Additionally, we should add `typescript` , `nodemon` , and `ts-node` , a TypeScript execution environment for Node.js. Run the following command:

```
npm install types/express @types/node typescript ts-node nodemon --save-dev
```

Next, we'll need to set up our `tsConfig.json` file, which provides instructions on how our TypeScript project should be configured. You can access the required TypeScript configuration for TypeGraphQL.

In `tsConfig.json` , we can specify options to compile our `.ts` files as well as the root files for our project. Whenever we run the `tsc` command, in our case, `npm run build` , the compiler will check this file first for special instructions, then proceed with compilation.

To create our `tsConfig.json` file, we can run the `tsc --init` command, which creates a new config file with many defaults and comments, which we've left out for brevity. Now, our `tsConfig` file looks like the following code block:

```
        "lib": ["dom", "es2016", "esnext.asynciterable"],
        "sourceMap": true,
        "emitDecoratorMetadata": true,
        "strict": false,
        "experimentalDecorators": true,
        "outDir": "dist",
        "rootDir": "app",
        "baseUrl": ".",
        "paths": {
            "*": [
                "node_modules/*",
                "app/types/*"
            ]
        }
    },
    "include": [
        "app/**/*", "./app/**/*.ts", "./app/**/*.tsx"
    ]
}
```

Note: Detailed interpretations and meanings of these configuration options can be found in the documentation.

Setting up our Apollo Server

Now, we can go ahead and set up our Apollo Server with the `apollo-server-express` package we installed earlier. Before doing so, we'll create a new app directory in our project directory. The contents of the directory should look like the image below:

The contents of our app folder

In our `server.js` file, we'll set up our Apollo Server with Express. The contents of the file with all the imports should look like the code below:

```
import { ApolloServer } from 'apollo-server-express';
import { ApolloServerPluginLandingPageGraphQLPlayground } from 'apollo-
server-core';
import Express from 'express';
import 'reflect-metadata';
import { buildSchema } from 'type-graphql';
import { connect } from 'mongoose';

import { UserResolver } from './resolvers/User';
import { ProductResolver } from './resolvers/Product';
import { CategoriesResolver } from './resolvers/Categories';
import { CartResolver } from './resolvers/Cart';
import { OrderResolver } from './resolvers/Order';

const main = async () => {
  const schema = await buildSchema({
    resolvers: [
      CategoriesResolver,
      ProductResolver,
```

The `buildSchema` package from TypeGraphQL allows us to build our schema from TypeGraphQL's definition. The usual signature of the `buildSchema` method is as follows:

```
const schema = await buildSchema({
  resolvers: [Resolver],
});
```

In the code above, we're importing our resolvers from the `app/resolver` folder and passing them into the array of the `resolvers` field inside the function definition. The `emitSchemaFile` field allows us to spit out our GraphQL schema into a `schema.gql` file when we run the `npm run build-tsc` command. `schema.gql` looks like the following:

```
  returnAllCart: [Cart!]!
  returnAllOrder: [Order!]!
}

"""The User model"""
type User {
  id: ID!
  username: String!

  email: String!
  cart_id: String!
  cart: Cart!
}

input UserInput {
  username: String!
  email: String!
  cart_id: ID!
}
```

TypeGraphQL database schema fields

The content inside of our `schema.gql` file is based on the schema fields for our different database entities, stored in the `entities` folder. Let's take a look at the contents in these files:

```
  [x: string]: any;
  @Field(() => ID)
  id: number;

  @Field()
  @Property({ required: true })
  username: String;

  @Field()
  @Property({ required: true })
  email: String;

  @Field((_type) => String)
  @Property({ ref: Cart, required: true })
  cart_id: Ref<Cart>;
}


export const UserModel = getModelForClass(User);
```

In the files above, we import `ObjectType` , `Field` , `ID` , and `Int` from `type-graphql` . The `Field` decorator is used to declare the class properties that should be mapped to the GraphQL fields. It is also used to collect metadata from the TypeScript reflection system. The `ObjectType` decorator marks the class as the `GraphQLObjectType` from `graphql-js` .

Additionally, we're importing both the `Property` decorator and the `getModelForClass` method from the `@typegoose/typegoose` package. The `Property` decorator is used for setting properties in a class, without which is just a type and will not be in the final model.

`Int` and `ID` are aliases for three basic GraphQL scalars, and the `getModelForClass` method is used to get a model for a given class. Lastly, we import `Refs` from the `types.ts` file in the app folder and `ObjectId` from MongoDB:

```
// app/types.ts
import { ObjectId } from 'mongodb';

export type Ref<T> = T | ObjectId;
```

The type `Ref<T>` is the type used for references. It also comes with typeguards for validating these references.

TypeGraphQL resolvers and input types

Go ahead and create a new folder called `resolver`, which will contain another folder called `types`. In `types`, we'll add the types for our different resolver inputs. The input files are shown below:

```
// app/resolvers/types/category-input.ts
import { InputType, Field } from 'type-graphql';
import { Length } from 'class-validator';
import { Categories } from '../../entities/Categories';

@InputType()
export class CategoriesInput implements Partial<Categories> {
  @Field()
  name: string;

  @Field()
  @Length(1, 255)
  description: String;
}



// app/resolvers/types/product-input.ts
import { InputType, Field } from 'type-graphql';
import { Length } from 'class-validator';
```

In the files above, we import the `InputType` and `Field` decorators from `type-graphql`. The `inputType` decorator is used by TypeGraphQL to automatically validate our inputs and arguments based on their definitions.

We are using the `class-validator` library for field-level validation. Note that TypeGraphQL has built-in support for argument and input validation based on this library.

Next, let's examine the resolvers for these inputs and entities. The content for the category resolver in the `categories.ts` file is shown below:

```
// app/resolvers/Categories.ts
import { Resolver, Mutation, Arg, Query } from 'type-graphql';
import { Categories, CategoriesModel } from '../entities/Categories';
import { CategoriesInput } from './types/category-input';

@Resolver()
export class CategoriesResolver {
  @Query((_returns) => Categories, { nullable: false })
  async returnSingleCategory(@Arg('id') id: string) {
    return await CategoriesModel.findById({ _id: id });
  }

  @Query(() => [Categories])
  async returnAllCategories() {
    return await CategoriesModel.find();
  }

  @Mutation(() => Categories)
  async createCategory(
```

This resolver performs basic CRUD operations using the `Resolver`, `Mutation`, `Arg`, and `Query` decorators from `type-graphql`. We're also importing the input types to be used for the mutation field. For the product resolver file, we have the following:

```
        price,
        category_id,
      })
    ).save();
    return product;
  }

  @Mutation(() => Boolean)
  async deleteProduct(@Arg('id') id: string) {
    await ProductModel.deleteOne({ id });
    return true;
  }

  @FieldResolver((_type) => Categories)
  async category(@Root() product: Product): Promise<Categories> {
    console.log(product, 'product!');
    return (await CategoriesModel.findById(product._doc.category_id))!;
  }
}
```

The product resolver above contains a field resolver decorator for relational entity data. In our case, the product schema has a `category-id` field for fetching details about a particular category, which we have to resolve by fetching that data from another node in our data graph.

Running our Apollo application

To start our application, we'll run `npm run build-ts`, which compiles our code, then `npm start`, which starts our server. Note that TypeScript catches any compile-time errors when we build our code with the `tsc` compiler:

Output after running the commands

When we're done, we can navigate to the GraphQL Playground at `http://localhost:3333/graphql` to test our API. Now, let's create a new category by running the following mutation:

```
mutation {
  createCategory(data: {
    name: "T-Shirts",
    description: "This is an awesome brand from LogRocket"
  }){
    name
    description
    id
  }
}
```

Creating a new category

To get a category by its ID, run the following query:

Returning a single category by ID

You'll see more details about the API's capabilities when you click on the **Schema** tab in the Playground:

```
input OrderInput {
  user_id: String!
  payde: Boolean!
  date: DateTime!
}

type Product {
  id: ID!
  name: String!
  description: String!
  color: String!
  stock: Int!
  price: Int!
  category_id: String!
  category: Categories!
}

input ProductInput {
  name: String!
```

To learn more, we can test the queries and the mutations in the schema tab shown above.

Conclusion

The main purpose of TypeGraphQL is to create GraphQL types based on TypeScript classes. TypeScript makes writing class-based OOP code intuitive. It provides us with classes, interfaces, and more out of the box, which then afford us the opportunity to properly structure our code in a reusable manner, making it easy to maintain and scale.

TypeGraphQL has led to the creation of tools and libraries that make it easier and faster to write applications that meet these expectations. TypeScript greatly benefits our productivity and experience as engineers.

By combining both TypeScript features and the benefits of GraphQL with the TypeGraphQL library, we are able to build resilient and strongly typed APIs that fulfill our needs in terms of maintenance, technical debt down the line, and so on.

As a final note, it would be great to explore other advanced guides and features in the documentation to learn more about other aspects not covered in this tutorial. Thanks for reading, and don’t forget to grab the entire source code used in this tutorial on GitHub.

Monitor failed and slow GraphQL requests in production

While GraphQL has some features for debugging requests and responses, making sure GraphQL reliably serves resources to your production app is where things get tougher. If you’re interested in ensuring network requests to the backend or third party services are successful, try LogRocket.https://logrocket.com/signup/

LogRocket is like a DVR for web and mobile apps, recording literally everything that happens on your site. Instead of guessing why problems happen, you can aggregate and report on problematic GraphQL requests to quickly understand the root cause. In addition, you can track Apollo client state and inspect GraphQL queries' key-value pairs.

Writing a lot of TypeScript? Watch the recording of our recent TypeScript meetup to learn about writing more readable code.

TypeScript brings type safety to JavaScript. There can be a tension between type safety and readable code. Watch the recording for a deep dive on some new features of TypeScript 4.4.

LogRocket instruments your app to record baseline performance timings such as page load time, time to first byte, slow network requests, and also logs Redux, NgRx, and Vuex actions/state. Start monitoring for free.

Alexander Nnakwue Follow

Software engineer. React, Node.js, Python, and other developer tools and libraries.

#graphql #typescript

8 Replies to "Integrating TypeScript with GraphQL"

Hubert Ursua Says: Reply
April 15, 2020 at 6:22 pm

Please update the type-graphql link to the new one (https://typegraphql.com/).

The old .ml URL is now compromised (see https://github.com/MichalLytek/type-graphql/issues/596).

> **Matt Angelosanto** Says: Reply
> April 16, 2020 at 7:53 am
>
> Thanks for pointing that out, all set.

Coder Says: Reply
June 13, 2020 at 6:38 pm

This is no longer working, just getting a whole bunch of errors when I download from github and try to compile it after running npm install.

Alexander Nnakwue Says: Reply
June 15, 2020 at 8:13 am

Hello @coder, can you paste the error you are getting here, so I can assist you with it?

John Says:
July 26, 2020 at 9:52 am
Reply

What is the __doc that has been added to entities?
Why is it required? I tried to upgrade to more current versions of the libraries and it no longer compiles.
Do you have any updates to this source?

Erfan Says:
January 2, 2021 at 4:14 am
Reply

is anyone have the source code ?

Jack Says:
March 7, 2021 at 9:27 pm
Reply

why @typegoose/typegoose is not a devDependancy? Instead of dependancy.

Giacomo Agent Says:
March 25, 2021 at 5:53 pm
Reply

where can I find github repo url for this project?

Leave a Reply