Newline Part 1

NODE

Node# Node is a JavaScript runtime environment that can run on different platforms (Mac, Windows, Linux, etc.). What this means is JavaScript (which was originally created to run inside a web browser) can now be run on any computer as a web server.

Node was originally released in 2009 by Ryan Dahl as a response to how slow web servers were at the time. This is because most web servers would block the I/O (Input/Output) task (e.g. reading from the file system or accessing the network) which will lower throughput. Node changed this model by making all I/O tasks non-blocking and asynchronous. Non-blocking, for example, just means a request from another interaction can be processed without waiting for the prior interaction request to finish. This allowed web servers to serve countless requests concurrently.

Non-blocking I/O# Here's an example taken from the main Node website in comparing code between the synchronous blocking state and the asynchronous non-blocking state.

This example covers the use of the Node File System (fs module) which allows us to work with the file system in our computer.

```
const fs = require("fs");
const data = fs.readFileSync("/file.md");
moreWork();
```

The file system module is included by stating require('fs'). The readFileSync() method is used to read files on the computer with which we've stated we want to read the markdown file named file.md. The readFileSync() method is synchronous so the moreWork() function will only run after the entire file is read in the process. Since the readFileSync() method blocks moreWork() from running, this is an example of synchronous blocking code.

As we've mentioned, Node allows I/O taks to be non-blocking and asynchronous and as a result provides asynchronous forms for all file system operations. Here's an attempt to read the file.md file and run the moreWork() function in an asynchronous setting:

```
const fs = require("fs");
let data;
fs.readFile("/file.md", (err, res) => {
  if (err) throw err;
  data = res;
});
moreWork();
```

The readFile() method is asynchronous and allows the use of a callback function. The callback function won't run until the former function, readFile(), is complete. In this case, the readFile() method doesn't block code since the moreWork() function will be run while the file is being read. Whenever the reading of the file is complete, the callback function will then run.

The ability to have the moreWork() function run alongside the reading of a file, in the asynchronous example above, was a primary design choice in Node to allow for higher throughput. Callback functions, promises, and the use of the async/await syntax are some of the ways in which Node allows us to conduct and use asynchronous functions.

Node and npm# If you've built any web application within a Node environment, you've already come across something known as npm. npm, short for Node Package Manager, is two things:

An online repository for the publishing of open-source Node Projects. A command-line utility for interacting with the npm repository. The ecosystem of third-party tools that can easily be installed in a Node application makes Node a very rich ecosystem. This also ties in with how we build applications in Node by introducing just the tools and libraries we need in an app.

One awesome benefit of using Node is that Node makes capable the building of universal JavaScript applications (often also known as isomorphic JavaScript). These are applications that have JavaScript both on the client and the server which is exactly what we'll be doing in our course!

Node is built against modern versions of V8 (Google's JavaScript and WebAssembly engine) which helps ensure Node stays mostly up to date with the latest ECMA JavaScript syntax. Node already supports a vast majority of ES6 features which can be seen in the node.green website.

Lets Create a Node Server!

we will create a package.json file which contains all scripts, dependencies and metadata.

package.json holds an object, and requires name and version inside!

`npm install express` adds dependenicies into package.json.

inside the `server` folder : `mkdir src && mv index.js src/`

```
const EXPRESS = require('express');
// we are requiring express module
const APP = EXPRESS();
//then creating app to express. this app is the server instance
const PORT = 9000;
// we are creating a port to communicate through

APP.get('/', (req, res) => res.send('server is up and running!') )

// now our express has get method whioch takes 2 arguments. the route
which is '/'here.
// and request, response tuple which defines when a request is made to
route, how to behave
```

```
APP.listen(PORT);

// here we are listening the server throughh desired and designated port.

console.log(`[APP]: http://localhost:${PORT}`)

// and for sanity check we are console logging what is happening
```

then in the terminal `node index.js`

`[APP]: http://localhost:9000` is ther terminal hayda and on localhost:9000, it tells me `server is up and running`

automating reloading with nodemon

its a hot-reload tool for express. `npm install nodemon -D`

cool we are done, except creating script in package.json file so we can run cli arguments.

in `package.json`, adding these :

```
    "scripts": {
        "start": "nodemon src/index.js"
    }
```

and in terminal `npm run start` will activate the nodemon.

now making changee in index.js file will automatically fire the nodemon and it will hot-reload the entire server ! cool!

adding typescript `npm install -D typescript ts-node`

then in root hayda, we cerate tsconfig.json

```
{
    "compilerOptions": {
        "target": "ES6",
        "module": "commonjs",
        "rootDir": "./src",
        "outDir": "./build",
        "esModuleInterop": true,
        "strict": true
    }
}
```

target# We'll declare the target option which specifies the target JavaScript version the compiler will output. Here we'll declare a target output of es6 since Node supports a vast majority of ES6 features.

"target": "es6", module# We'll declare the module option which refers to the module manager to be used in the compiled JavaScript output. Since CommonJS is the standard in Node, we'll state commonjs as the module option.

"module": "commonjs", rootDir# To specify the location of files for where we want to declare TypeScript code, we'll use the rootDir option and give a value of src/ to say we want our compiler to compile the Typescript code in the src/ folder.

"rootDir": "./src", outDir# We can use the outDir option to specify where we'd want to output the compiled code when we attempt to compile our entire TypeScript project into JavaScript. We'll dictate that we'll want this output code to be in a folder called build/.

"outDir": "./build", esModuleInterop# To help compile our CommonJS modules in compliance with ES6 modules, we'll need to introduce the esModuleInterop option and give it a value of true.

"esModuleInterop": true, strict# Finally, we'll apply the strict option which enables a series of strict type checking options such as noImplicitAny, noImplicitThis, strictNullChecks, and so on.

"strict": true The tsconfig.json file of our server project in its entirety will look like the following:

In order to use ts in orchestration with other libraries, they are also expected to have dynamic typing. but unfortunately that is not the case like express and node. So wee need to take action here so we will use declaration files.

so we will use gh repo `definitelytyped`

we need to download typescript declaration files for the libraries we use which are node and express js

`npm install -D @types/node @types/express` this will automatically fetch the declaration files from the gh repo and add to dev dependencies in package.json.

now we can use typescript, we will rename the index.js to index.ts

now in index file, instead of using require, we will use import that is being supported since es6.

Static Typing# The headline feature of TypeScript is static typing. Instead of having our variables be inferred as numbers, we can statically annotate the type of our variables as number with the syntax : number.

```
server/src/index.ts
const one: number = 1;
const two: number = 2;
TypeScript allows us to use and annotate many different basic types.

const three: boolean = false;

const three: string = "one";

const three: null = null;

const three: undefined = undefined;
```

```
const three: any = {};
```

There are other basic types as well such as the array type, the enum type, the void type, etc. We'll investigate many of these different types as we proceed through the course.

When we explicitly define the type of variable, we have to provide a value that matches that type. The any type in TypeScript is unique since it allows us to define a variable with any type. Variables with the any type don't give us the capability TypeScript provides and should be used sparingly.

to run the code, in the package.json, we change the index.js to index.ts file as following: so that it wont nag and say there is no js file in the src/ folder

```
    "scripts": {
        "start": "nodemon src/index.ts"
    }
```

`npm run start` working flawlessly!

beware of the nodemon output in the terminal `starting ts-node src/index.ts` 😀 our ts file is being used!

now sourcebase is ready but it is not over yet since we did not yield any javascript file yet. lets do it!

in order to finish compilation, we will write another script in the package.json file.

`"build": "tsc -p ./"` name of the command will be build and it will invoke tsc, which takes 2 arguments -p is same as --project and ./ means file is in here!

now we exit the terminal and will build `npm run build`

voila! a new folder 'build' is created and index.js it put into it per tsconfig.json dictates

note: we will nearly only use build script only before deploying the app. kitchen belongs to the typescript

ES Linting

linting: code checking for potential errors. ts and js is shining with eslint

in order to use it, we install the vscode extension and also install some dependencies to use it

`npm install -D eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin`

then in root directory, we will create a `.eslintrc.json` file

in the file we wrote :

```
{
    "parser": "@typescript-eslint/parser",
```

```
    "parserOptions": {
        "ecmaVersion": 2019,
        "sourceType": "module"
    },
    "extends": [
        "plugin:@typescript-eslint/recommended"],
    "env": {
        "node":true},
    "rules": {
        "indent": "off",
        "@typescript-eslint": "off",
        "@typescript-eslint/explicit-function-return-type": "off"
    }
}
```

not understanding these is not a great deal now.

so enabling eslint requires to go settings.json in vscode and adding the following :

```
"eslint.validate": [
  "javascript",
  "javascriptreact",
  { "language": "typescript", "autoFix": true },
  { "language": "typescriptreact", "autoFix": true }
]
```

then in the ts file, we get a eslint warning saying number type is inferred, do not write. bcs simple bindings are inferred easily, no need to write const one : number =1;.

now in the typescript file, we have req and res. we used res but not req. in order to suppress the error

message, we change `req res` to `_req res` so error is suppressed.

we built a very powerful working environment using VScode Intellisense + TypeScript + EsLint!

Mock Listing!

in our src folder, we create listings.ts file.

in this file we will have an array which holds different js objects. These objects contain properties like title, image, adress, price, numberofguests, etc. mimicking airbnb. total of 4 hayda.

In typescript, in complex data structures like array, we can specify union type using `\` symbol which means this hayda either can hold this or that type of data.

we can force the array to hold only specific type of data for each member. how ? there are two ways :

1. create `alias`
2. create `interface`

for example lets create an interface to force our array to a specific type of data.

above the const listings, we create its interface and calling it `Listing`, singular.

we add all properties of each listing in the listings and declare their types explicitly.

```
interface Listing{
    id: number;
    title: string;
    image: string;
    address: string
    price: number;
    numOfGuests: number;
    numOfBeds: number;
    numOfBaths:number;
    rating: number;
}
```

so, below, in listings, we cannot assign id a string or price to string etc. so how do we apply these?

well, since we created an object, now Listing is a type of object, of listing.

now we created this blueprint, we will force listings to obey to rules we define by explicitly assigingn type `Listing` to it.

```
export const listings : Listing[] =  [
    {
      id: "001",
      title:
        "Clean and fully furnished apartment. 5 min away from CN Tower",
    /// more code
```

> Listing[] => Listing denoting this is an array form factor

> anything before it, const listing in this example, should be compatible with it

note here : we could have written const listings: string[] as well but that would mean that this is a string array so all the members have to hold the type of string. But this is not the case so TypeScript would be angry. similarly, we could have said const listings :number[] and that would mean it is a number array , that moment strings would yell. you get the point.

Get and Post Express Routes

We will create two different routes in index.ts.

A listing route and delete listing route

but since we exported the listings, now it is time to import it into index.ts so we can refer to them.

`import {listings} from './listings';`

```
APP.get('/listings', (_req, res) => {
    return res.send(listings);
})
```

so, we creted a new route, which is /listings. we are not expecting to use request so suppressed it with _.

for response, we returned the response by sending all the listings we have.

and since this is a .get() method, obviously its a GET method.

now when we run the server and go to localhost:9000/listings endpoint, we are seeing our listings.

cool.

now lets create another endpoint to delete a listing from our array

for requesting data, we use get method. but to send or modify the server we use Post method.

in this case since we are deleting array member, we will use post method.

for this purpose, we will install a middleware so that it can parse the body.

this middleware is called `body parser`.

`npm install -D body-parser`

and then we will install its type declaration file and also add it to dev dependencies.

`npm install -D @types/body-parser`

then we will import this into our file `import bodyParser from 'body-parser';`

then we can utilize it .

`APP.use(bodyParser.json())` this will listen and parse the json returns from the server

now we will add a post request

```
APP.post('/delete', (req, res) => {
    const id: string = req.body.id;

    for (let i = 0; i < listings.length; i++) {
        if (listings[i].id === id) {
            return res.send(listings.splice(i, 1));

        }
    }
```

```
    return res.send('failed to delete listing!');
});
```

- we created an endpoint called delete
- this request will return a Json object with a body
- body.id will be set to const id. and to avoid problems, we strongly typed string.
- iterating over the list to see whethet provided id is in the list
- if so, we spliced this i and 1 element, which is the item itself and return it to us and remove from array
- else, we say failed to delete listing since there is no listing with that id.

now we will fire up the server and make the post request

```
curl -X POST http://localhost:9000/delete -H 'Content-Type: application/json' -d '{"id":"001"}'
```

- -X flag shows method of request
- followed by url
- -H flag shows header in this case it is application/json
- -d flag is the data body.

so , to this url, make a Post request, of type app/json and request body is id:001

this curl returned us with the desired post that is spliced from the list :

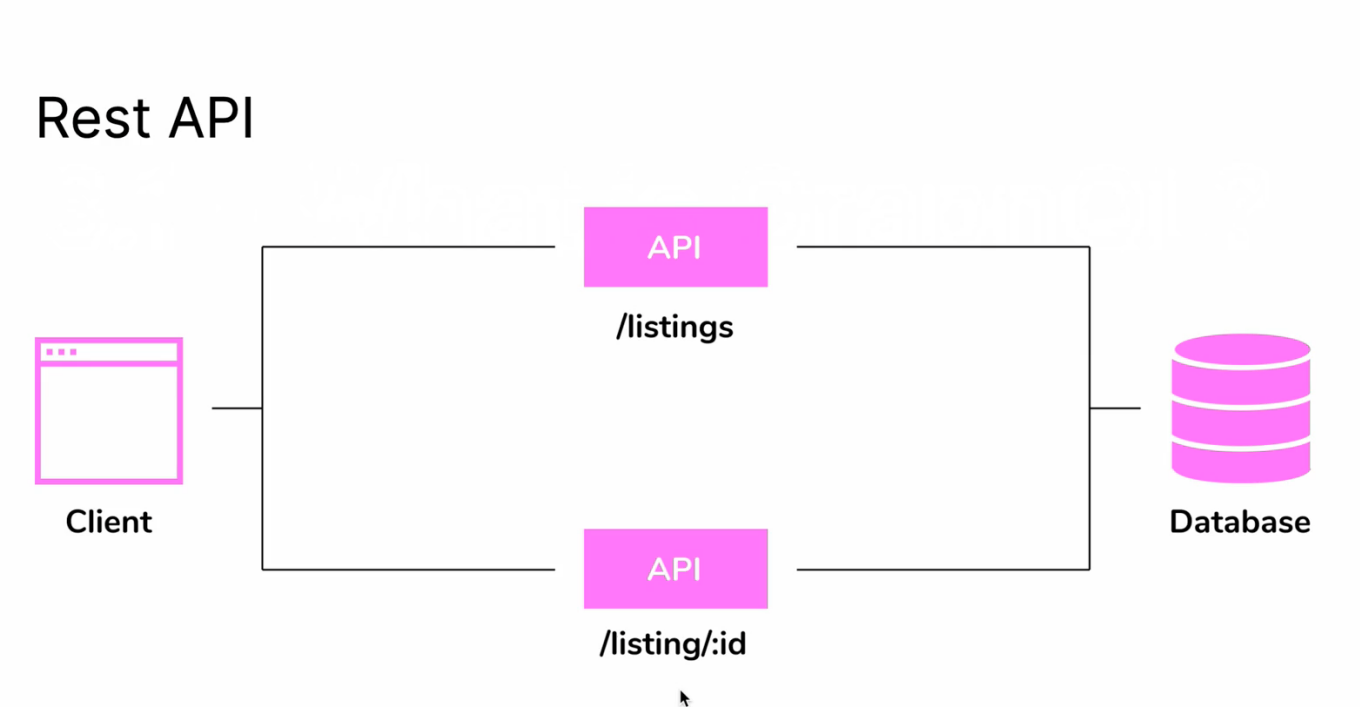
```
[{"id":"001","title":"Clean and fully furnished apartment. 5 min away from CN Tower","image":"https://res.cloudinary.com/tiny-house/image/upload/v1560641352/mock/Toronto/toronto-listing-1_exv0tf.jpg","address":"3210 Scotchmere Dr W, Toronto, ON, CA","price":10000,"numOfGuests":2,"numOfBeds":1,"numOfBaths":2,"rating":5}
]
```

upon refreshing the page, we wont be able to see a listing with the id of 001 anymore since it is spliced!

same curl retrned this : `failed to delete listing!`

GraphQL API

1. Rest API



in traditional rest api, there are various endpoints for get, put etc for interacting with the database.

fetch user data and listings data for certain user: /user/:id => to fetc certain user data

/user:id/listings => to fetch listings for certain user

so making the first request, for user id and then the second request for listings entailed to that user

so far so good, becasue there is not much requests to make anyways.

however with graphQL, clients asks `exactly what he needs!` and server returns this exact data

> graphQL is tecnology agnostic. can be used with any backend or frontend library/ framework, including RoR!

an example for syntax? there you go!

```
query User($id: !ID){
    user(id: $id){
        id
        name
        listings{
            id
            title
        }
    }
}
```

so we query a user with id and we can specify which areas we want to be returned

it is like `SELECT id , name FROM user WHERE id = x;`

> ! means this is a required field!

so what is the benefit of graphQL?

> intuitive -> client specifies exactly what data it needs

> performant -> no useless data neets to ebe transferred

> typed -> every field in the schema is type-defined

> self-documented

> consist of a single endpoint

for using grapql in languages like python, java, ruby, js, there are different client libraries.

in our case we will be using Apollo client

in classical rest api, we get aaal the json as response. this creates lots of littering around and data usage. for example, querying api.github.com/repos/octocat/Hello-World url returns tons and tons of information whereas we wanted only the description!!!

graphql comes into play right here actually. we specify what EXACTLY we need and get what we asked for.

using public api of github, we queried :

```
query{
  repository(owner:"octocat" ,name:"Hello-World") {
    description

  }
}
```

which returned

```
{
  "data": {
    "repository": {
      "description": "My first repository on GitHub!"
    }
  }
}
```

only what we wanted for.

lets say we want to have the following information :

- description of the repo
- a specific issue number , in this case is 348
- first 5 comments in this issue
- names of the people who commented.

in REST API model, we had to make 3 different rrequests, one to general api, one to api/user/issues and one to api/user/issues/comment. in graphQL, single query is enough.

```
query{
  repository(owner:"octocat", name: "Hello-World"){
    description
    issue(number:348){
      title
      comments(first:5){
        edges{
          node{
            bodyText
            author{
              login
            }
          }
        }
      }
    }
  }
}
```

so in the repo, we want description, issue and comments and these are direct children of the body and each have their own children

GraphQL Core Concepts

every graphql api come with a schema which contains all the possible data we can request

the Schema Resolvers, turn the data into schema.

the most basic component of graphql schjema is object types.

in our example, since we defined a data type, Listing, this object type is usable by graphql

```
type Listing{
    id : ID!
    title:String!
    address:String!
    price:Int!
}
```

and we write the type so there is type checking. this language is called graphql schema langugage

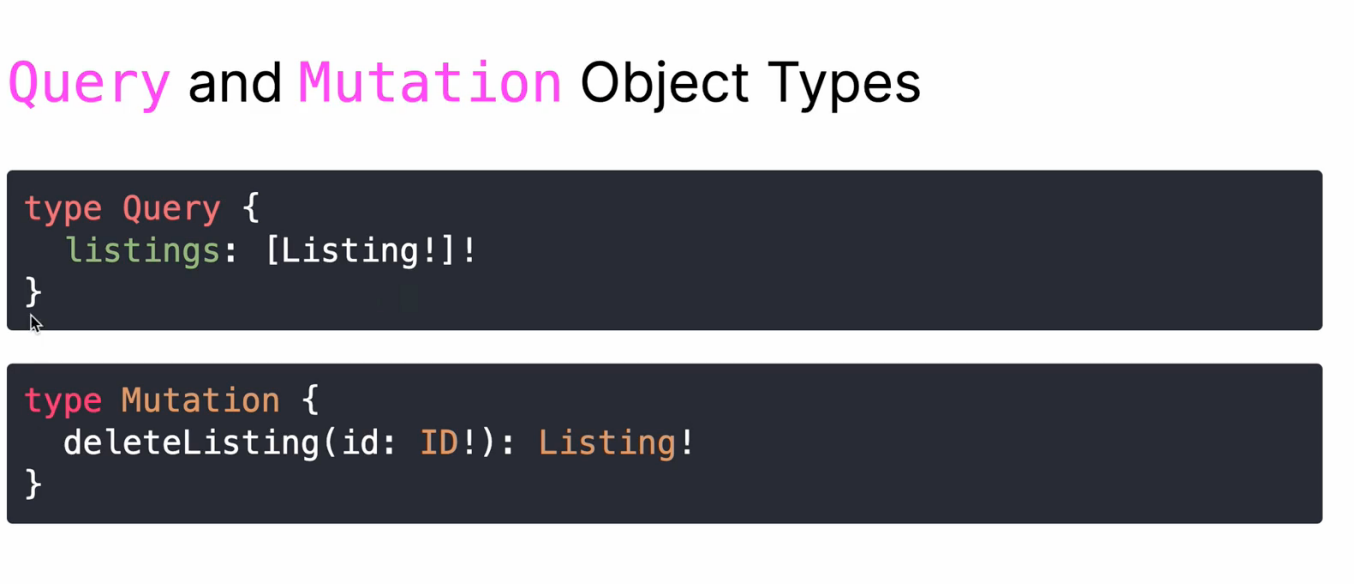
we can build one to one, one to many, many to one relations with graphql

There are 2 special types in graphql schema :

> Query Mutation

Every graphql schema must have Query type and may or may not have a mutation type.

mutation is for mutating the data, like removing one node



these are entry points of querying.

scalar types:

these are defalt types that do not harbor no more children :

- field:Boolean!
- field:Int!
- field:Float!

- field:String!
- field:ID!

and also we have `enum types` which is restricted to a defined set of allowed values like this:

```
enum ListingType{
    HOUSE
    APARTMENT
}
```

this says our listing type can only be either house or an apartment.

so in graphql, objects, scalar and enums are only types we can define. within the query, we can combine these elemenets. for example we can query for a object and set the return value for a list of another object.

```
type Listing{
    bookings:[Booking]
}
```

as such.

but wait, why did not we add ! for booking? because ! means essenttially these values are required and Cannot be of null value. However, sometimes we can query for somethings than can be null. for example a particular user in airbnb maybe did not book any place in last 3 months. so his booing list will return empty. if threr is such possibility, we do not add ! for booking.

here, for example,

```
type Query {
  listings: [Listing!]!
}
```

```
type Mutation {
  deleteListing(id: ID!): Listing!
}
```

in first query , we say we expect a non-empty listing (outer !) and each return value should be compatible with the Listing structure and type(inner !)

in graphql, we can pass arguments to the fields as well. Why? becasue arguments are essentially like functions that return values.

```
type Mutation{
    createListing(
```

```
        id:ID!
        title:String!
        address:String!
        price:String!
    ); Listing!
}
```

but for convention, thhere is another way to do it in graphql, which is using `input` type.

instead of above, here is better

```
type Mutation{
    createListing(input: createListingInput!):Listing!
}
```

in thi case, create listing field takes a nun-nullable input argument of type create listing input.

and how do these fields resolve to return data?

ta taa using `resolvers`

a resolver can be a function that returns an array of listings

```
Query:{
    listings:()=>{
        return listings;
    },
}
```

notice that listings are defined elsewhere!

mutation is similar:

Root resolvers!

```
Mutation:{
    deleteListing(obj,args,ctx)=>{
        for(let i= 0,i<listings.length, i++){
            if(listings[i].id ===args.id){
                return listings.splice(i,1)[0];
            }
        }
    },
}
```

Field resolvers:

```
Listing:{
    id:(obj) =>obj.id,
    title(obj) =>obj.title
}
```

so id resolves to id of the object here.

so what are these obects, args etc?

A graphql resolver always take 4 positional arguments.

> obj : object returned from the parent resolver args : arguments provided to the field context: value provided to every resolver info: info about the execution state of the query

Apollo Server Core Concepts

OK. GraphQL is awesome but how we will use this tech for our JS heavy product?

We will use Apollo Server for creating our API, this is its job. Build Graphql API

Apollo enables us to connect our graphql schema to a server or even work witout serverless environments

lets install graphql and apollo server, shall we?

`npm i apollo-server-express`, this is a type-script hayda, so we do not need tgo install extra type-checking packages for type-check. since graphql lib is a peer to apollo, we also need to install it

`npm install graphql`

but graphql library does not have its own type definitions. so we need to add another plugin for that. `npm install -D @types/graphql`

in order to start using apollo server, we need to import it just like we imported express server.

`import ( ApolloServer ) from 'apollo-server-express';`

then we create our server `const SERVER = new ApolloServer();`

after removing many unneeded imports and endpoints from our index.ts, since we will use graphql to create them again, here how it looks now before starting filling them:

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';

const APP = express();
const PORT  =9000;
const SERVER = new ApolloServer();
SERVER.applyMiddleware({APP, path: '/api'});

APP.listen(PORT)
```

```
console.log(`[app]: http://localhost:${PORT}`);
```

ApolloServer expects a schema and type-definition as argment. now we will create them using graphhql and pass these as argument to apolloserver class instance

This sschema will be built in a new file tho 😃

in the /src folder, we now have a `graphql.ts` file.

our new apollo express server needs object type and schema from graphhql right?

so we need to create them. and to do so ,

```
import {GraphQLObjectType, GraphQLSchema} from 'graphql';
```

where GraphQLObjectType is the most basic component of the graph schema wihch can be used to represent to aaaall types from basic, to even mutation and custom types.

GraphQLSchema is used to create schema by passing graphql object types

this is the prototype of our graph api:

```
import {GraphQLObjectType,GraphQLString GraphQLSchema} from 'graphql';

const query = new GraphQLObjectType({
    name: 'Query',
    fields:{
        hello: {
            type:GraphQLString,
            resolve:()=>'Hello from the query!'
        }
    }
});

const mutation = new GraphQLObjectType({
    name:'Mutation',
    fields:{
        hello: {
            type:GraphQLString,
            resolve:()=>'Hello from the mutation!'
        }
    }
});

export const SCHEMA = new GraphQLSchema({query, mutation});
```

- after importing the required imports,
- created 2 class instances from graphql schema one for query one for muation.

- also created a schema and exported it. this schema takes object types defined above as arguments.
- each object has to have a name, so we can refer to them and also fields.
- for each field, we need to specify a type and a resolver for that type.
- in order to do that, we used GraphQLString since our type is of type string
- and resolve is invoke function that describes when this api call is made.
- now it is read to be passed into apollo server.
- we import this schema to index.ts and pass schema into apollo server middleware as argment

```
const app = express();
const PORT  =9000;
const SERVER = new ApolloServer({schema});
SERVER.applyMiddleware({app, path: '/api'});
```

- now in the terminal, `npm run start` and in `localhost:9000/api` path, voila! we have a surprise!

in this path dwells an inteactie IDE to interact with our own api.

so witout having to use postman or curl, we will be able to test our api and schema here for all possible scenarios before deployment. amazing.

lets query our own API 😄

our api responds to the query :

```
query{
  hello
}
```

with

```
{
  "data": {
    "hello": "Hello from the query!"
  }
}
```

just as expected!

Querying and mutating listings data with GraphQL

Alright in order to use our listings, we need to create a custom object type in graphql.ts

```
const Listing = new GraphQLObjectType({
    name: 'Listing',
    fields:{
```

```
        id:{type: GraphQLNonNull(GraphQLID)},
        title:{type: GraphQLNonNull(GraphQLString)},
        image:{ type: GraphQLNonNull(GraphQLString)},
        address:{type: GraphQLNonNull(GraphQLString)},
        price:{type:GraphQLNonNull(GraphQLInt)},
        numOfGuests:{type: GraphQLNonNull(GraphQLInt)},
        numOfBeds:{type: GraphQLNonNull(GraphQLInt)},
        numOfBaths:{type: GraphQLNonNull(GraphQLInt)},
        rating:{type: GraphQLNonNull(GraphQLInt)}
    }
});
```

note that we import GraphQLInt, GraphQLString etc at the top level import. We always want to define the types of the fields of our query, never want them to be null. we import GraphQLNonNull to ensure that. and wrapping each marker with that!. So these fields must force these types!

The modified version of query object is like the following:

```
const query = new GraphQLObjectType({
    name: 'Query',
    fields:{
        listings:{
            type: GraphQLNonNull(GraphQLList(GraphQLNonNull(Listing))),
            resolve: ()=>{
                return listings
            }
        }
    }
})
```

- new const with the type of GraphQLObjectType
- name is Query
- it holds some fields:
- the field is listing. what listing that we defined earlier holds, is the field for this object
- as type, it is a Listing type. for to be parsed, we write the type, which is GraphQLList type.
- now, neither the list, nor the return of field should retun null.
- thus first the listing then the List itself is wrapped with nonNull wrapper.
- and as resolve, this returns the list of listings.

the mutation object is a little bit more complex

```
const mutation = new GraphQLObjectType({
    name:'Mutation',
    fields:{
        deleteListing: {
            type:GraphQLNonNull(Listing),
            args:{
```

```
                id:{type:GraphQLNonNull(GraphQLID)}
            },
            resolve:(_root,{id} )=>{
                for (let i = 0; i < array.length; i++) {
                    if(listings[i].id === id){
                        return listings.splice(i,1)[0];
                    }

                }
                throw new Error('failed to delete the listing');

            }
        }
    }
});
```

- type is a nonnull listing,
- in the arguments part, args, we look for specific ID through which the query will be executed
- the return type is , listing.
- we wont be using root argument so prefixed with _.
- then look for the specific list item to splice it
- splice returns a list with 1 element inside, which is the first element of array.
- in order to access it, we used [0].
- if there is not an item matching with the id, an error is thrown.

lets try the api, shall we 😄

in the playground, simply

```
query{
    listings{
        id
        title
    }
}
```

returns :

```
{
  "data": {
    "listings": [
      {
        "id": "001",
        "title": "Clean and fully furnished apartment. 5 min away from CN
Tower"
      },
      {
        "id": "002",
```

```
        "title": "Luxurious home with private pool"
      },
      {
        "id": "003",
        "title": "Single bedroom located in the heart of downtown San
Fransisco"
      },
      {
        "id": "004",
        "title": "Come To Kefelikoy! Earth, hidden from crowds"
      }
    ]
  }
}
```

as for deleting an item :

```
 mutation{
 deleteListing(id: "001"){
  id
  title
  }
}
```

I want the listing with the ID of 001 to be deleted and upon deletion, return me the following values: id and title

```
{
  "data": {
    "deleteListing": {
      "id": "001",
      "title": "Clean and fully furnished apartment. 5 min away from CN
Tower"
    }
  }
}
```

But this is very verbose and with big apps, it can be very tiring to write and debug. Now we will write it again but this time using GraphQL schema language.

to do that, inside the source folder we create a graphql folder. inside of which we will create 3 ts files : typeDef.ts, resolvers.ts and index.ts

in the type definitions file we will begin to define our type definitions.

```
import {gql} from 'apollo-server-express';
```

```
export const typeDefs = gql`
    type Listing{
        id: ID!
        title: String!
        image: String!
        address: String!
        price:Int!
        numOfGuests: Int!
        numOfBeds: Int!
        numOfBaths: Int!
        rating: Int!
    }

    type Query{
        listings: [Listing!]!
    }

    type Mutation{
        deleteListing(id: ID!): Listing!
    }
`;
```

that easy with the graphql schema language. no more verbose wrapping , no more go inside json key again and again. here we import gql, and wrapping all the schema with 2 back-ticks. inside we created 3 types: listing itself, this is the object, Query method and Mutation method. beware that deletelisting takes id as argument.

> also ! means nonnull.!!!

now we will write the resolvers in resolvers.ts file

```
import { listings } from "../listings";
import{IResolvers} from 'apollo-server-express';

export const resolvers:IResolvers = {
  Query: {
    listings: ()=> {
      return listings;
    },
  },
  Mutation: {
    deleteListing: (_root: undefined , {id}:{id: string}) => {
      for (let i = 0; i < listings.length; i++) {
        if (listings[i].id === id) {
          return listings.splice(i, 1)[0];
        }
      }
      throw new Error("failed to delete listing");
    },
  },
```

```
};
```

IResolvers is a tool that we use to specify the type of the function so we import it.

in `deleteListing: (_root: undefined , {id}:{id: string}) =>`, we do not use root so hide it and type is undefined.

as for id, we destructire it as {id} and we know it is serialized into type string. so we defined it. rest is regular if check.

in src/graphql/index.ts, we export these:

```
export * from './resolvers';
export * from './typeDefs';
```

and in the index.ts file in the src file we will import them directly from ./graphql folder.

now we dont need graphql.ts file and can remove it since we declared all using graphql schema language and also we should remove the schema import from the src/index.ts file.

in terminal we run the server this time again.

we can start using querying , works like charm.

however, whenever we restart our server ,all the mutated data comes back. this is beause we hardcoded our list.

we need a database to make it smarter. come to the next chapter!

MongoDB and Persistent Data

after creating the mongodb atlas account, creating cluster and adding a fake data, we can install mongodb driver from terminal `npm install mongodb`

in order to be able to use it with typescript, we need to install also declarative file which forces typecheck. `npm install -D @types/mongodb`

in src folder, we will create a database folder where we will set up our db connection settinfs.

note that database config files should never be commited to source code!!

here is the config for mongodb server

```
import {MongoClient} from 'mongodb';


const user = username;
const password = password
const cluster = clustername
```

```
const dbname ='test_listing';
const url =
`mongodb+srv://${user}:${password}@${cluster}.mongodb.net/${dbname}?
retryWrites=true&w=majority`

export const connectDatabase = async() =>{
  const client = await MongoClient.connect(url, {
      useNewUrlParser:true,
      useUnifiedTopology:true
    });

   const db = client.db('main');

   return{
    listings:db.collection('test_listings')
   }

};
```

now there needs to be a communication between mongo connection and express local server.

in src/index.ts oh boy, we seriosly modified the document!

```
import express, {Application} from 'express';
import { ApolloServer } from 'apollo-server-express';
import { resolvers,typeDefs} from './graphql';
import {connectDatabase} from  './database';

const PORT  =9000;

const mount = async (app: Application) => {  //async function, bcs
connection can take some time
    const db = await connectDatabase();  // from mongoDB
    const SERVER = new ApolloServer({typeDefs,resolvers, context: ()=>
{db}}); // here apollo server took context as argument and db is passed in
    SERVER.applyMiddleware({app, path: '/api'});  // these 3 lines are
moved to here.

    app.listen(PORT)

    console.log(`[app]: http://localhost:${PORT}`);

    const listings = await db.listings.find({}).toArray();  // .find is
mongodb function and ({}) means find all
    console.log(listings)
};

mount(express());  // finally, this mount function takes the express as
parameter thanks to importing of {Application} from express
```

now lets give it a try:) npm run start!

I had manually entered a single list item into database before. this connection returned this object back

```
  {
    _id: 5ffdf5694b03fd0811e76bfb,
    title: 'Clean and fully furnished apartment  - minutes from metro
driveway',
    address: 'kefelikoy Istanbul Cuhmuriyet Mahallesio'
  }
]
```

we have some problems in the code tho. for example many of the types of arrays or etc are type of `any` which nullifies the typescript benefits. we need to fix it! so we need to make some type definitions. in order to do it, we create src/lib folder and inside, types.ts file.

```
import {Collection,ObjectId} from 'mongodb';

export interface Listing{
    _id:ObjectId;  //start with _ bcs its primary key so mongodb can
recognize it type is ObjectId, a mongo thing.
    title:string;
    image:string;
    address:string;
    price:number;
    numOfGuests: number;
    numOfBeds: number;
    numOfBaths: number;
    rating:number;
}

export interface Database{
    listings:Collection<Listing>;  // assigning all listings a type of
Collection, a mongodb thing. and explicitly assigining type of Listing
that we created just above. so anywhere in codebase we use listings, it
will hold Collection<Listing> type.

}
```

what is typescript generics tho?

generics is a tool exist in manyt other languagaes that create a type to use later.

I think it is like `typedef struct` in C language.

notes that this is type of X. we could have used any mesela ama this would not be helpful.

gotta check interfaces and generics of typescript.

after finishing Database file in lib folder, we import it into index.ts file in database folder.

since connectDatabase function is asynchronous, it wxpects a promise. so this new generic, Database, will be wrapped with Promise.

`export const connectDatabase = async(): Promise<Database>=>{..`

like so.

meaning that this connectDatabase will return a promise and when resolved, the return value will be an object of Database.

We need to create environment variables so we can store pass and usernames securely not hardcoded!

in node, we call these using `process.env.variable`

we will use `dotenv` library for managing this.

`npm install -D dotenv @types/dotenv`

in the root directory, we create `.env` file

here we add these varuables

after setting these, we go to src/index.ts and AT THE TOP! of the imports, we import htis file.

`require('dotenv').config();`

or in package.json, in start script we can add this.

```
scripts": {
        "start": "nodemon -r dotenv/config src/index.ts"
}
```

now in places we passed port, username, pass, we will change to process.env.X, process.env.y

now our database/index.ts file is like following:

```
import {MongoClient} from 'mongodb';
import {Database} from '../lib/types';

const url =
`mongodb+srv://${process.env.DB_USER}:${process.env.DB_USER_PASSWORD}@${pr
ocess.env.DB_CLUSTER}.mongodb.net/${process.env.DB_NAME}?
retryWrites=true&w=majority`

export const connectDatabase = async(): Promise<Database>=>{
  const client = await MongoClient.connect(url, {
      useNewUrlParser:true,
      useUnifiedTopology:true
    });

   const db = client.db('main');
```

```
    return{
     listings:db.collection('test_listings')
    }
};
```

everything is working , perfect!

Seeding Mock Data to MongoDB

we nned to create a new seed file in a new temp folder in the root folder. Since this seed.ts file is only to be used for development, we create it outside of src folder, as a standalone folder.

so we create a seed function and also require our environment variables to connect to the database.

this seed function will try to send our mock data to database , and if faild, it will catch the error. this is, ofc, an async function.

```
const seed = async () =>{
    try{
        console.log('[seed]: Running...')
    }
    catch
    {
        throw new Error('Failed to seed database')
    }
}
```

also we need our connectDatabase function to connect, so we import it and assign it to a variable.

```
const db = await connectDatabase();
```

now we will copy our mock listings and use these.

then we create a script in package.json, run the script and voila, our listings are seeded to our database.

```
require("dotenv").config();
import {connectDatabase} from '../src/database';
import{Listing} from '../src/lib/types';
import {ObjectId} from 'mongodb';

const seed = async () =>{
    try{
        console.log('[seed]: Running...');
        const db = await connectDatabase();
        const listings: Listing[]=[
    {
      _id: new ObjectId(),
```

```
      title:
        "Clean and fully furnished apartment. 5 min away from CN Tower",
      image:
        "https://res.cloudinary.com/tiny-
house/image/upload/v1560641352/mock/Toronto/toronto-listing-1_exv0tf.jpg",
      address: "3210 Scotchmere Dr W, Toronto, ON, CA",
      price: 10000,
      numOfGuests: 2,
      numOfBeds: 1,
      numOfBaths: 2,
      rating: 5
    },
    {
      _id: new ObjectId(),
      title: "Luxurious home with private pool",
      image:
        "https://res.cloudinary.com/tiny-
house/image/upload/v1560645376/mock/Los%20Angeles/los-angeles-listing-
1_aikhx7.jpg",
      address:
        "100 Hollywood Hills Dr, Los Angeles, California",
      price: 15000,
      numOfGuests: 2,
      numOfBeds: 1,
      numOfBaths: 1,
      rating: 4
    },
    {
      _id: new ObjectId(),
      title:
        "Come To Kefelikoy! Earth, hidden from crowds",
      image:

"https://imgcy.trivago.com/c_lfill,d_dummy.jpeg,e_sharpen:60,f_auto,h_450,
q_auto,w_450/itemimages/96/95/96959_v6.jpeg",
      address: "Kefelikoy, Sariyer",
      price: 443000,
      numOfGuests: 2,
      numOfBeds: 2,
      numOfBaths: 4,
      rating: 5
    }
        ];
        for(const listing of listings){
            await db.listings.insertOne(listing);
    }
        console.log('[seed]: success')
    }
    catch
    {
        throw new Error('Failed to seed database')
    }
}
```

```
seed();
```

this is the final version of the seed file, notice that we changed the id field to _id so it fits with our interface. Also it was string but we called new Objectid() so that unique id is created.

Modulirazing resolvers

data can get bigger so we may need to modularize the resolvers , as best practice not as a must.

under graphql, we crate resolvers>Listing>index.ts and copy all the content of resolvers.ts into this file and then delete the resolvers.ts file. For each requirement, we will create a diffrent mapping under resolvers folder to modularize it. to do that, we will use lodash library.

```
npm i lodash.merge
```

```
npm install @types/lodash.merge -D
```

in resolvers folder, we create index.ts file.

```
import merge from 'lodash/merge';
import {listingResolvers} from './Listing';

export const resolvers = merge(listingResolvers);
```

we changed the name resolve to listingResolvers in the index.ts file in Listings directory.

Now in future, apart from Listings, if we craete other resolvers like booking, users, etc, we can follow the similar steps to modilarize all the resolvers and merge them easily using lodash library