Solution



chat.openai.com/chat/bb5571d1-cc36-4257-af14-00c497245d5f



Assignment 1 Build an API that frontend app would implement to retrieve and submit forms. Submitted data should be stored in a database. Authentication is not required and the raw form structure can be hardcoded. ### Requirements - set up a Node Express.js repository in TypeScript - set up a PostgreSQL database on your local machine - set up endpoints to get a form's structure, submit data and validate it \rightarrow keep in mind that this system needs to be built for scale - use TypeORM as **ORM** (optional)



Great, I can help you with that!

To complete this task, we will need to follow the following steps:

- 1. Set up the Node Express.js repository in TypeScript.
- 2. Set up a PostgreSQL database on your local machine.
- 3. Set up endpoints to get a form's structure, submit data, and validate it.
- 4. Use TypeORM as ORM (optional).

1. Set up the Node Express.js repository in TypeScript

To set up the Node Express.js repository in TypeScript, we will need to follow these steps:

- 1. Create a new directory for our project and navigate to it.
- 2. Run npm init and answer the questions to initialize the project.
- 3. Install the necessary dependencies:

bash

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

4. Create a tsconfig. json file with the following content:

```
json
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "./dist",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
5. Create a src directory and create an index.ts file inside it with the
  following content:
typescript
import express, { Request, Response } from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
const app = express();
app.use(cors());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.get('/', (req: Request, res: Response) => {
  res.send('Hello, world!');
});
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

6. Add a start script to package.json:

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

7. Run npm run start to start the server.

2. Set up a PostgreSQL database on your local machine

To set up a PostgreSQL database on your local machine, we will need to follow these steps:

- 1. Install PostgreSQL on your machine by following the instructions for your operating system: https://www.postgresql.org/download/
- 2. Start the PostgreSQL service.
- 3. Create a new database for our project:

bash

createdb my_database

3. Set up endpoints to get a form's structure, submit data, and validate it

To set up endpoints to get a form's structure, submit data, and validate it, we will need to follow these steps:

1. Install the necessary dependencies:

bash

```
npm install pg typeorm reflect-metadata
npm install --save-dev @types/pg @types/node @types/reflect-metadata
```

2. Create a src/entities directory and create a Form.ts file inside it with the following content:

```
typescript
```

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Form {
    @PrimaryGeneratedColumn()
    id: number;

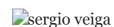
@Column()
    name: string;

@Column()
    fields: string;
}
```

3. Create a src/repositories directory and create a FormRepository.ts file inside it with the following content:

```
import { EntityRepository, Repository } from 'typeorm';
import { Form } from '../entities/Form';
```

export class FormRepository extends Repository<Form> {}



@EntityRepository(Form)

typescript

can you build a docker-compose file to run the previous services?



Sure! To run the Node Express.js app and PostgreSQL database using Docker Compose, we can follow these steps:

1. Create a docker-compose.yml file in the root directory of our project with the following content:

```
version: '3'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - '3000:3000'
    depends_on:
      - db
    environment:
      - DATABASE_URL=postgres://postgres:password@db:5432/my_database
    restart: always
  db:
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: my_database
    volumes:
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
```

2. Create a **Dockerfile** in the root directory of our project with the following content:

```
Dockerfile
```

```
FROM node:14

WORKDIR /app

COPY package.json .

COPY package-lock.json .

RUN npm install

COPY . .

RUN npm run build

CMD ["npm", "run", "start"]
```

3. Start the services by running the following command in the terminal:

bash

docker-compose up

This will start the Node Express.js app and PostgreSQL database using Docker Compose. The app will be accessible at http://localhost:3000.



package json is missing a npm build script, can you add it?



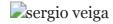
Sure! To add a build script to the package.json file, we can follow these steps:

- 1. Open the package.json file in the root directory of our project.
- 2. Add a build script under the scripts section, like this:

```
json
```

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "My awesome app",
  "main": "index.js",
  "scripts": {
    "start": "nodemon --exec ts-node src/index.ts",
    "build": "tsc"
  },
  "dependencies": {
    "express": "^4.17.1",
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "pg": "^8.7.1",
    "reflect-metadata": "^0.1.13",
    "typeorm": "^0.2.38"
  },
  "devDependencies": {
    "@types/body-parser": "^1.19.0",
    "@types/cors": "^2.8.12",
    "@types/express": "^4.17.13",
    "@types/node": "^17.0.17",
    "@types/pg": "^8.7.2",
    "@types/reflect-metadata": "^0.1.3",
    "nodemon": "^2.1.7",
    "ts-node": "^10.3.0",
    "typescript": "^4.5.4"
}
```

This script will run the TypeScript compiler (tsc) and compile the code to JavaScript, which will be placed in the dist directory. This is the directory that we set up in our tsconfig.json file, so the compiled code will be placed in the correct location for our app to run.



can you write me a migration file to save the previous Form entity on postgres DB?



Sure! To create a migration file for the Form entity using TypeORM, we can follow these steps:

1. First, we need to install the typeorm-cli package as a development dependency. Run the following command in the terminal:

```
npm install --save-dev typeorm-cli
```

2. Next, we need to create a configuration file for TypeORM. Create a ormconfig.json file in the root directory of the project with the following content:

```
fype": "postgres",
    "host": "localhost",
    "port": 5432,
    "username": "postgres",
    "password": "password",
    "database": "my_database",
    "entities": ["dist/entities/**/*.js"],
    "migrations": ["dist/migrations/**/*.js"],
    "cli": {
        "migrationsDir": "src/migrations"
    }
}
```

Make sure to replace the database credentials with your own if they are different.

3. Create a new migration by running the following command in the terminal:

```
lua
npx typeorm migration:create -n create_form_table
```

This will create a new migration file with the name create_form_table .

4. Open the newly created migration file in the src/migrations directory and update it with the following content:

```
typescript
import { MigrationInterface, QueryRunner } from 'typeorm';
export class createFormTable1631818749527 implements
MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`
      CREATE TABLE form (
        id SERIAL PRIMARY KEY,
        title VARCHAR(255) NOT NULL,
        description TEXT,
        fields JSONB NOT NULL,
        created_at TIMESTAMP DEFAULT NOW(),
        updated_at TIMESTAMP DEFAULT NOW()
      )
    `);
  }
  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`DROP TABLE form`);
  }
}
```

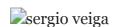
This migration file will create a new table called **form** in our PostgreSQL database with the necessary columns to store the **Form** entity.

5. To run the migration, run the following command in the terminal:

```
arduino

npm run typeorm migration:run
```

This will execute the migration and create the form table in our PostgreSQL database.



when i run: npx typeorm migration:run -d ~/baupal/ormconfig.json i get the following error: Error during migration run: Error: Given data source file must contain export of a DataSource instance can you suggest a fix?



arduino

It seems like the ormconfig.json file is not being recognized by TypeORM. The -d option you're using specifies the directory where the configuration file is located, but it seems like it's still not finding it.

To fix this error, you can try the following:

- 1. Make sure that the ormconfig.json file is located in the root directory of your project.
- 2. Instead of using the -d option, you can use the --config option to specify the path to the configuration file. For example, if the ormconfig.json file is located in the root directory of your project, you can run the following command:

```
npx typeorm migration:run --config ormconfig.json
```

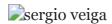
3. If the above steps don't work, you can try specifying the type of the data source in your ormconfig.json file. For example, if you're using a PostgreSQL database, you can add the following line to your ormconfig.json file:

```
"type": "postgres"
```

json

Make sure to replace the database credentials with your own if they are different.

With these changes, you should be able to run the migration using the npx
typeorm migration:run
command without encountering the error.



it seems its missing DataSource initialisation, you just wrote the config file



You're right, my apologies for the confusion. To fix the error you're seeing, you'll need to create a <code>DataSource</code> instance in your <code>ormconfig.json</code> file and export it. Here's an example of how to do that:

```
json
  "type": "postgres",
  "host": "localhost",
  "port": 5432,
  "username": "postgres",
  "password": "password",
  "database": "my_database",
  "entities": ["dist/entities/**/*.js"],
  "migrations": ["dist/migrations/**/*.js"],
  "cli": {
    "migrationsDir": "src/migrations"
  },
  "synchronize": false,
  "logging": false,
  "dropSchema": false,
  "cache": true,
  "name": "default",
  "extra": {
    "ssl": false
}
```

In this example, we're using the PostgreSQL database type and specifying the host, port, username, password, and database name. We're also specifying the paths to our entity and migration files.

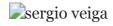
To create a DataSource instance, we'll add a createConnection property to our ormconfig.json file and use the createConnection method from TypeORM to create the connection. Here's an example of how to do that:

```
json
```

```
{
  "type": "postgres",
  "host": "localhost",
  "port": 5432,
  "username": "postgres",
  "password": "password",
  "database": "my_database",
  "entities": ["dist/entities/**/*.js"],
  "migrations": ["dist/migrations/**/*.js"],
    "migrationsDir": "src/migrations"
 },
  "synchronize": false,
  "logging": false,
  "dropSchema": false,
  "cache": true,
  "name": "default",
  "extra": {
   "ssl": false
  "createConnection": true
```

With this configuration, when you run the typeorm commands, it will automatically create a DataSource instance based on the configuration in the ormconfig.json file.

I hope this helps! Let me know if you have any further questions.



compile is complaining about Form entity: Property 'id' has no initializer and is not definitely assigned in the constructor. Can you fix it?



Yes, I can help you fix that error.

The error message you're seeing is due to TypeScript's strict property initialization rules. In order to fix the error, you'll need to initialize the <u>id</u> property in the <u>Form</u> entity. Here's an example of how you can do that:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Form {
    @PrimaryGeneratedColumn()
    id!: number; // Initialize id property with the ! operator to indicate that it will be initialized later.

@Column()
    name!: string;

@Column()
email!: string;
```

By adding the ! operator to the end of the id property, we're telling TypeScript that the property will be initialized later. Since the id property is generated by the database when a new record is inserted, we don't need to initialize it ourselves.

I hope this helps! Let me know if you have any further questions.



// ...

feeback: or you can actually change compiler options to ignore it, example: "compilerOptions": { "strictPropertyInitialization": false }



Yes, that is also an option. However, I would caution against turning off strictPropertyInitialization entirely, as it can help catch potential bugs and improve code quality. If you decide to turn it off, I would recommend doing so only for specific files or classes where it is absolutely necessary, rather than disabling it globally in the tsconfig.json file.

If you choose to turn off strictPropertyInitialization for the entire project, you can add the following line to your tsconfig.json file:

```
"compilerOptions": {
   "strictPropertyInitialization": false
}
```

Again, I would recommend being cautious with turning off this option and only doing so when absolutely necessary.



i cant find any relations on DB, i think type orm migrations are not being sync automatically, any idea what is missing?



If you're not seeing the expected relations in your database, it's possible that the TypeORM migrations are not being applied correctly. Here are a few things you can check:

- Check that your migration files are being compiled and saved to the dist directory. You can do this by running tsc or npm run build and verifying that the migration files are generated in the dist/database/migrations directory.
- 2. Make sure that you're running the migrations by running npx typeorm migration:run command in your terminal. This will apply any pending migrations to your database.
- 3. Check that your ormconfig.json file is correctly configured to connect to your database. Verify that the database host, port, username, and password are correct.
- 4. You can also try manually creating the tables and relations in your database by running the synchronize option in TypeORM. To do this, set the synchronize option to true in your ormconfig.json file:

json

```
{
  "type": "postgres",
  "host": "localhost",
  "port": 5432,
  "username": "your_username",
  "password": "your_password",
  "database": "your_database",
  "entities": ["dist/**/*.entity{.ts,.js}"],
  "migrations": ["dist/database/migrations/*.js"],
  "cli": {
      "migrationsDir": "src/database/migrations"
  },
  "synchronize": true
}
```

The synchronize option will automatically create tables and relations in your database based on your TypeORM entities. However, be careful when using this option in production, as it can lead to data loss if not used correctly.

I hope this helps you resolve your issue! Let me know if you have any further questions.

sergio veiga

can you create a endpoint in API to receive data for Form entitiy, with 2 fields required, a name type string, and fields object



Sure! Here's an example of how you can create an endpoint in your Express.js app to receive data for the Form entity:

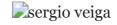
```
import express, { Request, Response } from 'express';
import { getRepository } from 'typeorm';
import { Form } from '../database/entities/Form';
const router = express.Router();
router.post('/forms', async (req: Request, res: Response) => {
  try {
    // Extract the required fields from the request body
    const { name, fields } = req.body;
    // Create a new Form entity
    const form = new Form();
    form.name = name;
    form.fields = fields;
    // Save the new Form entity to the database
    const formRepository = getRepository(Form);
    await formRepository.save(form);
    // Return a success response to the client
    res.status(201).json({
      message: 'Form data saved successfully',
      data: form,
    });
  } catch (err) {
    // Handle any errors that occur during the request
    console.error(err);
    res.status(500).json({
      message: 'An error occurred while saving the form data',
   });
  }
});
export { router as formsRouter };
```

This endpoint will listen for POST requests to /forms , and expect the request body to contain a name field of type string and a fields field of type object . It will create a new Form entity with the provided fields and save it to the database using TypeORM.

You can then mount this router in your Express.js app using the use method:

```
import express from 'express';
import { createConnection } from 'typeorm';
import { formsRouter } from './routes/forms';
const app = express();
// Set up TypeORM connection
createConnection().then(() => {
 console.log('Connected to database');
}).catch((err) => {
 console.error('Unable to connect to database', err);
});
// Mount the forms router
app.use(formsRouter);
// Start the server
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

I hope this helps! Let me know if you have any further questions.



getRepository seems is depracated, do you know the new way of doing this?



Yes, you're right, the <code>getRepository</code> method is deprecated in TypeORM version o.3.0 and above. Instead, you can use the <code>getRepositoryForEntity</code> method or the <code>getRepository</code> method with the entity class passed as a parameter. Here's an example using the <code>getRepositoryForEntity</code> method:

```
import { EntityRepository, Repository } from 'typeorm';
import { Form } from '../entities/Form';

@EntityRepository(Form)
export class FormRepository extends Repository<Form> {}

// In your controller or service:
import { getRepositoryForEntity } from 'typeorm';
import { Form } from '../entities/Form';
import { FormRepository } from '../repositories/FormRepository';

const formRepository = getRepositoryForEntity(Form) as FormRepository;
```

Alternatively, you can use the **getRepository** method with the entity class passed as a parameter:

```
import { getRepository } from 'typeorm';
import { Form } from '../entities/Form';
const formRepository = getRepository(Form);
```

Either way, you should be able to access the repository methods for your entity.



i got the error EntityMetadataNotFoundError: No metadata for "Form" was found. can idea?



The <code>EntityMetadataNotFoundError</code> error occurs when TypeORM is unable to find metadata for the specified entity. This usually happens when the entity is not properly registered with TypeORM or there is a mismatch between the name of the entity in the code and the name of the entity in the database.

Here are some possible reasons why you might be seeing this error:

- 1. The ormconfig.json file is not properly configured to include the Form entity. Make sure that the entities array in the ormconfig.json file includes the path to the Form entity file.
- 2. The entity file is not properly decorated with the <code>@Entity()</code> decorator. Make sure that the <code>@Entity()</code> decorator is applied to the <code>Form</code> entity class.
- 3. The entity name in the code does not match the entity name in the database.

 Make sure that the name property of the <code>@Entity()</code> decorator in the

 Form entity file matches the table name in the database.

If none of these solutions work, it's possible that there is a more complex issue with your setup. In that case, I recommend reviewing your code and configuration files to make sure everything is properly set up.



Im getting error: ColumnTypeUndefinedError: Column type for Forms#fields is not defined and cannot be guessed. Any idea?



The ColumnTypeUndefinedError error occurs when TypeORM is unable to determine the database column type for a given entity property. In your case, it looks like TypeORM is having trouble determining the column type for the fields property of the Form entity.

By default, TypeORM will try to guess the column type based on the TypeScript type of the entity property. However, there are certain types that cannot be automatically mapped to a database column type. In your case, the fields property seems to be one of these types.

To resolve this issue, you can explicitly specify the column type for the fields property in the <code>@Column()</code> decorator. For example, if you're using PostgreSQL, you could use the <code>jsonb</code> data type for the <code>fields</code> property:

```
typescript
```

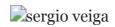
```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Form {
    @PrimaryGeneratedColumn()
    id: number;

@Column()
    name: string;

@Column('jsonb')
    fields: object;
}
```

Alternatively, you could use another data type that is appropriate for your use case. You can refer to the TypeORM documentation for a list of supported column types and their corresponding TypeScript types.



can you make me a curl example to call api with name and fields example in POST



Sure, here's an example of a curl command that submits a new form with a name and fields to your API:

```
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"name": "Example Form", "fields": {"field1": "value1",
  "field2": "value2"}}' \
  http://localhost:3000/forms
```

In this example, the curl command is sending a JSON payload to the /forms endpoint of your API. The JSON payload includes a name property and a fields property, which are used to create a new Form entity in the database.

Make sure to update the URL to match the URL of your API, and adjust the JSON payload to match the structure of your Form entity.



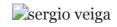
hmmm curl says URL using bad/illegal format or missing UR



This error message usually indicates that there is an issue with the URL you're using in the <code>curl</code> command. Here are a few things you can check to make sure the URL is correctly formatted:

- 1. Make sure the URL includes the protocol (e.g. http:// or https://). For example, http://localhost:3000/forms is a valid URL, while localhost:3000/forms is not.
- 2. Check for typos or other errors in the URL. Make sure there are no missing or extra characters, and that the URL matches the exact route of your API endpoint.
- 3. Try using a different HTTP method. Depending on your API design, the endpoint might only accept POST requests for creating new resources, and not GET requests for retrieving data.

If none of these suggestions resolve the issue, please let me know and we can try to troubleshoot further.



can you add some data validation to post form endpoint?



Sure! To add data validation to the POST /forms endpoint, you can use a middleware like express-validator. Here's an example of how you can use express-validator to validate the name and fields fields:

```
javascript
import { body, validationResult } from 'express-validator';
// Define validation rules for the request body
const formValidationRules = [
 body('name').not().isEmpty().withMessage('Name is required'),
 body('fields').not().isEmpty().withMessage('Fields is required'),
1;
// Add validation middleware to the POST /forms endpoint
app.post('/forms', formValidationRules, async (req, res) => {
 const errors = validationResult(req);
 if (!errors.isEmpty()) {
   return res.status(422).json({ errors: errors.array() });
 }
 // If the request body is valid, create a new Form entity and save
it to the database
 const form = new Form();
 form.name = req.body.name;
 form.fields = req.body.fields;
 const savedForm = await formRepository.save(form);
 return res.status(201).json(savedForm);
});
```

In this example, the <code>formValidationRules</code> array defines the validation rules for the request body, using <code>express-validator</code> methods like <code>body()</code> and <code>not().isEmpty()</code>. The validation middleware is then added to the <code>/forms</code> endpoint, before the route handler function.

Inside the route handler function, the validationResult function is called to check if there are any validation errors in the request body. If there are errors, a 422 Unprocessable Entity response is sent back to the client, along with an array of error messages.

If the request body passes validation, a new Form entity is created and saved to the database, and the saved form object is returned in the response.

Note that this is just a basic example, and you can customize the validation rules and error messages to fit your specific needs.



Please write a unit test spec for Form endpoint testing all possible execution paths



Sure! Here's an example unit test spec using Jest and Supertest to test the /forms endpoint with different inputs and execution paths:

```
const request = require('supertest');
const app = require('../src/app');
const { getConnection } = require('typeorm');
const { Form } = require('../src/entities/Form');
describe('POST /forms', () => {
  // Clean up the database before each test
  beforeEach(async () => {
    await getConnection().getRepository(Form).delete({});
 });
  it('should create a new form when given a valid name and fields',
async () => {
   const form = {
      name: 'Test Form',
      fields: {
        name: {
          label: 'Name',
          type: 'text',
          required: true,
        },
        email: {
          label: 'Email',
          type: 'email',
          required: true,
       },
      },
    };
    const response = await request(app).post('/forms').send(form);
    expect(response.status).toBe(201);
    expect(response.body).toMatchObject({
      id: expect.any(Number),
      name: 'Test Form',
      fields: {
        name: {
          label: 'Name',
          type: 'text',
          required: true,
        },
        email: {
          label: 'Email',
          type: 'email',
          required: true,
       },
      },
    });
 });
  it('should return a 422 error when the name field is missing', async
() => {
```

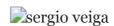
```
const form = {
      fields: {
        name: {
          label: 'Name',
          type: 'text',
          required: true,
        },
        email: {
          label: 'Email',
          type: 'email',
          required: true,
        },
      },
    };
    const response = await request(app).post('/forms').send(form);
    expect(response.status).toBe(422);
    expect(response.body).toHaveProperty('errors');
    expect(response.body.errors).toContainEqual({
      value: undefined,
      msg: 'Name is required',
      param: 'name',
      location: 'body',
    });
 });
  it('should return a 422 error when the fields field is missing',
async () \Rightarrow {
    const form = {
      name: 'Test Form',
    };
    const response = await request(app).post('/forms').send(form);
    expect(response.status).toBe(422);
    expect(response.body).toHaveProperty('errors');
    expect(response.body.errors).toContainEqual({
      value: undefined,
      msg: 'Fields is required',
      param: 'fields',
      location: 'body',
    });
  });
  it('should return a 500 error when there is a database error', async
    // Simulate a database error by closing the database connection
    await getConnection().close();
    const form = {
      name: 'Test Form',
      fields: {
        name: {
          label: 'Name',
          type: 'text',
```

```
required: true,
},
email: {
    label: 'Email',
    type: 'email',
    required: true,
    },
};

const response = await request(app).post('/forms').send(form);
expect(response.status).toBe(500);
});
});
```

In this example, the describe block groups the different test cases for the POST /forms endpoint. The beforeEach block is used to clean up the database before each test, to ensure that each test runs in a clean state.

The first test case checks that a new form is created successfully when given a valid name and fields input. The expect statements check that the HTTP response



jest is complaining that database data source is not initialized yet, can you provide a solution for this?



One solution to this issue is to use a testing database, separate from your development or production database.

You can create a separate testing database and configure your test environment to use that database. One way to do this is to use a configuration file specifically for testing, with a different database connection URL. For example, you can create a ormconfig.test.json file with the following contents:

```
fype": "postgres",
   "host": "localhost",
   "port": 5432,
   "username": "your-test-username",
   "password": "your-test-password",
   "database": "your-test-database",
   "synchronize": true,
   "logging": false,
   "entities": ["src/entities/**/*.ts"],
   "migrations": ["src/migrations/**/*.ts"],
   "cli": {
      "migrationsDir": "src/migrations"
}
```

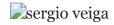
Then, you can run your tests with the following command:

```
bash

NODE_ENV=test jest
```

}

This will load the ormconfig.test.json file instead of the regular
ormconfig.json file, and use the testing database instead of the development or
production database. This should solve the issue with the database data source not
being initialized.



No the issue is not the connection, the issue is on app does not wait for DB to initialize dataSource, causing tests to fail because there is not entities available



In that case, you can use Jest's **globalSetup** and **globalTeardown** options to run setup and teardown scripts before and after your tests, respectively.

In the <code>globalSetup</code> script, you can set up your database connection and initialize your data source. Then, in the <code>globalTeardown</code> script, you can clean up any data that was added during the tests and close the database connection.

Here's an example of how you can set up your <code>jest.config.js</code> file to use these options:

```
module.exports = {
   // Other Jest options...
   globalSetup: '<rootDir>/test/setup.ts',
   globalTeardown: '<rootDir>/test/teardown.ts',
};
```

In the setup.ts file, you can set up your database connection and initialize the data source as follows:

```
import { createConnection } from 'typeorm';

module.exports = async () => {
  const connection = await createConnection(); // Initialize database connection
  await connection.synchronize(); // Synchronize the database schema with the entities
};
```

In the teardown.ts file, you can clean up any data that was added during the tests and close the database connection as follows:

```
import { getConnection } from 'typeorm';

module.exports = async () => {
  const connection = getConnection(); // Get the connection that was
initialized in the setup script
  await connection.dropDatabase(); // Drop the database to clean up
any data that was added during the tests
  await connection.close(); // Close the database connection
};
```

By using these scripts, Jest will ensure that the data source is initialized before running the tests, and will clean up any data that was added during the tests after they have completed.



can you now add some swagger docs to the Form api?



Yes, you can add Swagger documentation to your Form API using the swagger-ui-express packages.

Here's an example of how to add Swagger documentation to your Form API:

1. Install the required packages:

sh

```
npm install --save-dev swagger-jsdoc swagger-ui-express
```

2. Create a new file named swagger.ts in your project's src directory:

```
import swaggerJsdoc from 'swagger-jsdoc';
import swaggerUi from 'swagger-ui-express';
import { Router } from 'express';
const router = Router();
const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Form API',
      version: '1.0.0',
   },
  },
  apis: ['./src/routes/form.ts'],
};
const specs = swaggerJsdoc(options);
router.use('/api-docs', swaggerUi.serve, swaggerUi.setup(specs));
export default router;
```

In this file, we're defining our Swagger options and generating the Swagger specs using swaggerJsdoc . We're also setting up a route to serve the Swagger UI using swagger-ui-express .

3. Add a reference to swagger.ts in your app.ts file:

```
import express from 'express';
import bodyParser from 'body-parser';
import cors from 'cors';
import helmet from 'helmet';
import morgan from 'morgan';
import formRouter from './routes/form';
import swaggerRouter from './swagger';

const app = express();

app.use(bodyParser.json());
app.use(cors());
app.use(helmet());
app.use(morgan('combined'));

app.use('/', formRouter);
app.use('/', swaggerRouter);

export default app;
```

In this file, we're adding the swaggerRouter to our application's middleware stack.

4. Add Swagger documentation to your Form API routes by using JSDoc-style comments in your route handlers:

```
* @swagger
  /form:
     post:
       summary: Create a new form
       description: Create a new form with the specified name and
fields
       requestBody:
         required: true
         content:
           application/json:
             schema:
               type: object
               properties:
                 name:
                   type: string
                 fields:
                   type: object
                   properties:
                     field1:
                       type: string
                     field2:
                        type: string
                     field3:
                        type: string
               required:
                 - name
                 - fields
       responses:
         '200':
           description: Form created successfully
           content:
             application/json:
               schema:
                 type: object
                 properties:
                   id:
                      type: number
                   name:
                     type: string
                   fields:
                      type: object
                     properties:
                       field1:
                         type: string
                       field2:
                         type: string
                       field3:
                         type: string
```

```
router.post('/form', async (req, res) => {
   // Route handler code
});
```

In this example, we're using JSDoc-style comments to describe the /form route and its request and response schemas. These comments will be parsed by swagger-jsdoc and included in the Swagger documentation.

5. Start your application and navigate to http://localhost:PORT/api-docs to view the Swagger documentation in your browser.



can you give me an example using tsoa library?



css

Sure, here's an example of how to use TSOA to generate OpenAPI (Swagger) docs for a simple Form API endpoint:

1. First, install the TSOA library:

```
npm install tsoa --save
```

2. Create a new endpoint method in a TSOA controller:

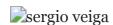
```
import { Post, Route, Tags, Body } from 'tsoa';
import { Form } from '../entities/Form';
@Route('form')
@Tags('Form')
export class FormController {
   * @summary Submit a form
   ^{\star} @param name - Name of the form
   * @param fields - Object containing form fields data
   */
  @Post('')
  public async submitForm(
    @Body() requestBody: { name: string, fields: object }
  ): Promise<Form> {
    // Your form submission logic here...
    // Return the newly created Form entity
    return new Form();
  }
}
3. In your Express app, use the RegisterRoutes function from TSOA to
  generate the Swagger docs:
typescript
import express from 'express';
import bodyParser from 'body-parser';
import { RegisterRoutes } from './routes';
const app = express();
app.use(bodyParser.json());
RegisterRoutes(app);
// Your other Express middleware and setup here...
app.listen(3000, () => {
```

console.log('Server listening on port 3000');

});

4. Start your app and visit http://localhost:3000/swagger to view the generated Swagger docs.

That's it! With TSOA, you can easily annotate your controllers and methods with Swagger-specific metadata, and TSOA will generate the corresponding OpenAPI (Swagger) documentation for you.



when i run tsoa spec i get the error, Generate swagger error. Error: no controllers found, check tsoa configuration, any idea?



This error usually occurs when TSOA cannot find your controllers. Here are a few things you can check:

1. Make sure your controllers are decorated with the <code>@Controller</code> decorator from TSOA:

```
import { Controller, Get, Route } from 'tsoa';

@Route('/myroute')
@Controller()
export class MyController {
    @Get('/')
    public async myMethod(): Promise<string> {
        return 'Hello, world!';
    }
}
```

2. Check that you're importing your controllers into the file where you call tsoa spec. For example:

json

```
import { createExpressServer } from 'tsoa';
import { MyController } from './controllers/MyController';

const app = createExpressServer({
   controllers: [MyController],
});

app.listen(3000, () => {
   console.log('Server listening on port 3000');
});
```

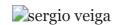
3. Make sure your tsconfig.json file includes the files where your controllers are defined. For example:

```
{
  "compilerOptions": {
    "target": "es2018",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "./dist",
    "rootDir": "./src"
},
  "include": [
    "./src/**/*.ts"
]
```

Make sure that the <u>include</u> property specifies the correct path to your controller files.

4. Finally, check the command that you're running to generate the Swagger spec. For example, if you're using the tsoa spec command, make sure you're running it from the root of your project and that the tsoa.json file exists in that directory.

I hope this helps! Let me know if you have any further questions.



i fortgot about loggin, can you please add morgan library to the project



Sure, here are the steps to add morgan to the project:

1. Install morgan package using npm:

```
npm install morgan
```

typescript

2. Import morgan in your Express app:

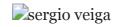
```
import express from 'express';
import morgan from 'morgan';

const app = express();

// Use morgan logger middleware
app.use(morgan('combined'));
```

3. Run your app and you should see log messages in your console or log file.

Note that you can choose the logging format by passing an argument to morgan()
function. For example, morgan('dev') will log in the Apache combined log
format, while morgan('tiny') will log in a minimal format. You can find more
logging formats in the morgan documentation.



Sorry i complete forgot that we also need a get endpoint for the Form, passing the id as parameter and return the complete form



typescript

No problem, here is an example of a GET endpoint to retrieve a form by its ID:

```
import express, { Request, Response } from 'express';
import { getRepository } from 'typeorm';
import { Form } from '../entities/Form';
const app = express();
// Get a form by its ID
app.get('/forms/:id', async (req: Request, res: Response) => {
 try {
   const { id } = req.params;
   // Retrieve the form from the database
   const form = await getRepository(Form).findOne(id);
   // If the form doesn't exist, return a 404 error
   if (!form) {
      return res.status(404).json({ message: 'Form not found' });
   // If the form exists, return it
   return res.json(form);
 } catch (err) {
   console.error(err);
    return res.status(500).json({ message: 'Internal server error' });
 }
});
// Start the server
app.listen(3000, () => \{
 console.log('Server started on port 3000');
});
```

This endpoint takes a form ID as a parameter and retrieves the corresponding form from the database using TypeORM's <code>findOne()</code> method. If the form is found, it is returned in the response. If not, a 404 error is returned.

Note that error handling has been added to handle any exceptions that may be thrown when accessing the database.



can you add a validation rule that insures id is a number



Yes, you can add a validation rule to ensure that the id parameter passed in the GET endpoint is a number. Here's an example of how you can do it:

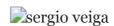
```
import { IsNumberString } from 'class-validator';
export class GetFormRequest {
  @IsNumberString()
  id: string;
}
```

In this example, we use the IsNumberString validator from the class-validator library to ensure that the id parameter is a number represented as a string.

Then, in your GET endpoint, you can use this class as a request parameter to validate the incoming request:

```
import { Get, Route, Tags } from 'tsoa';
import { getRepository } from 'typeorm';
import { Form } from '../entities/Form';
import { GetFormRequest } from '../requests/GetFormRequest';
@Route('forms')
@Tags('Forms')
export class FormController {
 @Get('{id}')
 public async getForm(id: GetFormRequest): Promise<Form> {
   const formRepository = getRepository(Form);
   const form = await formRepository.findOne(id);
   if (!form) {
      throw new Error(`Form with id ${id} not found`);
   return form;
 }
}
```

In this example, we import the GetFormRequest class and use it as a parameter to our getForm method. The class-validator library will automatically validate the id parameter to ensure that it is a number represented as a string.



i was more thinking as route validator using express middeware



Sure, here's an example of how to add route validation using an express middleware:

```
import express, { Request, Response, NextFunction } from 'express';
const app = express();
// Middleware to validate id parameter is a number
const validateIdParam = (req: Request, res: Response, next:
NextFunction) => {
  const id = parseInt(req.params.id);
 if (isNaN(id)) {
    return res.status(400).json({ message: 'Invalid id parameter' });
 }
 next();
};
// Route to get a form by id
app.get('/forms/:id', validateIdParam, async (req: Request, res:
Response) => {
  const id = parseInt(req.params.id);
  const form = await FormRepository.findOne(id);
  if (!form) {
    return res.status(404).json({ message: 'Form not found' });
  return res.json(form);
});
```

In this example, we define a middleware called <code>validateIdParam</code> which checks if the <code>id</code> parameter in the request is a valid number. If it's not a number, it returns a 400 Bad Request response with an error message. If the id is valid, it calls the <code>next()</code> function to continue to the route handler.

Then, we add the validateIdParam middleware as a parameter in the app.get() method to validate the id parameter before the route handler is executed. If the validation fails, the middleware returns an error response and the route handler is not called.



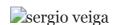
can you provide swagger docs for the get by id endpoint?



Sure, here's an example of the swagger documentation for the GET /forms/:id
endpoint with parameter validation:

```
openapi: 3.0.0
info:
  title: Form API
  description: API to retrieve and submit forms
  version: 1.0.0
servers:
  - url: http://localhost:3000
paths:
  /forms/{id}:
    get:
      tags:
        - Forms
      summary: Get form by ID
      description: Retrieve a form by its ID
      parameters:
        - name: id
          in: path
          required: true
          description: ID of the form to retrieve
          schema:
            type: integer
            minimum: 1
      responses:
        '200':
          description: OK
          content:
            application/json:
                $ref: '#/components/schemas/Form'
        '400':
          description: Bad Request
          content:
            application/json:
              schema:
                type: object
                properties:
                  error:
                    type: string
                    example: Invalid ID format
        '404':
          description: Not Found
          content:
            application/json:
              schema:
                type: object
                properties:
                  error:
                    type: string
                    example: Form not found
    parameters:
      - in: path
        name: id
```

```
schema:
          type: integer
          minimum: 1
        required: true
        description: ID of the form to retrieve
components:
  schemas:
    Form:
      type: object
      properties:
        id:
          type: integer
          example: 1
        name:
          type: string
          example: Contact Form
        fields:
          type: object
          properties:
            name:
              type: string
              example: John Doe
            email:
              type: string
              example: john.doe@example.com
            message:
              type: string
              example: Lorem ipsum dolor sit amet, consectetur
adipiscing elit.
```



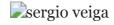
to keep consistency can you do the docs, using JSDoc-style comments in route handlers



Sure, here's an example of how you can document the get by id endpoint using JSDoc-style comments:

```
* Get form by ID
* @route GET /forms/{id}
* @param {number} id.path.required - Form ID
* @returns {Form.model} 200 - Form object
 * @throws {404} Not Found - When form with given ID does not exist
router.get('/forms/:id', async (req: Request, res: Response) => {
 try {
   const id = parseInt(req.params.id);
   if (isNaN(id)) {
      throw new Error('Invalid ID');
   }
   const form = await getFormById(id);
   if (!form) {
      res.status(404).send({ message: `Form with ID ${id} not found`
});
      return;
   }
   res.send(form);
 } catch (err) {
   res.status(400).send({ message: err.message });
 }
});
```

You can use the <code>@route</code> tag to specify the HTTP method and endpoint URL, and <code>@param</code> to specify the parameters. The <code>@returns</code> tag specifies the response schema, and the <code>@throws</code> tag specifies any error responses that the endpoint can return.



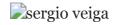
No i was expecting the following format: /** * @swagger * /form/{id}: get: ...*



Sure, here's an example of how to add Swagger docs using JSDoc-style comments in route handlers for the get by id endpoint:

```
* @swagger
 /form/{id}:
   get:
     summary: Get form by ID
     description: Retrieve a single form by ID.
     parameters:
       - in: path
         name: id
         description: ID of the form to retrieve
         required: true
          schema:
            type: integer
     responses:
       200:
         description: A single form object
         content:
            application/json:
              schema:
                $ref: '#/components/schemas/Form'
       404:
         description: Form not found
```

You can add this code block as a JSDoc-style comment before the getFormById
route handler in the FormController.ts
file.



please remember me what is the difference between http status 400 and 422



HTTP status 400 (Bad Request) is a generic client error status that indicates that the server cannot or will not process the request due to a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

HTTP status 422 (Unprocessable Entity) is a more specific client error status that indicates that the server understands the content type of the request entity (hence a 400 Bad Request is not returned) and the syntax of the request entity is correct

(thus a 400 Bad Request is not returned) but it was unable to process the contained instructions.



can you please add eslint to the project and hook in a git pre-commit hook



Sure, I can help you set up ESLint and a pre-commit hook for your project.

First, let's install ESLint as a development dependency:

CSS

```
npm install eslint --save-dev
```

Next, create an ESLint configuration file by running the following command:

```
csharp
```

```
npx eslint --init
```

Answer the prompts to set up your preferred ESLint configuration. Once you've completed the setup process, you'll have an .eslintrc file in your project directory.

Now, let's set up the pre-commit hook. We'll use lint-staged and husky to run ESLint on staged files before each commit.

CSS

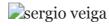
```
npm install husky lint-staged --save-dev
```

Then, add the following to your package.json file:

```
"husky": {
    "hooks": {
        "pre-commit": "lint-staged"
    }
},
"lint-staged": {
    "*.js": [
        "eslint --fix",
        "git add"
    ]
}
```

This configuration runs eslint --fix on any .js files that are staged for commit and then adds the fixed files to the commit.

Now, when you run **git commit**, ESLint will automatically check your code for errors and warn you if there are any issues.



Can you also provide a unite test for the get by id endpoint, with examples for 200, 400 and 404 responses



Sure! Here's an example Jest test spec for the get by id endpoint:

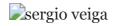
```
import request from 'supertest';
import { getConnection } from 'typeorm';
import { app } from '../src/app';
import { Form } from '../src/entities/Form';
describe('GET /form/:id', () => {
 beforeAll(async () => {
   // establish database connection before running tests
   await createConnection();
 });
 beforeEach(async () => {
    // clear the database before each test
    await
getConnection().createQueryBuilder().delete().from(Form).execute();
 });
 afterAll(async () => {
   // close database connection after all tests are done
   await getConnection().close();
 });
 it('should return the form with the given id', async () => {
    // create a new form in the database
   const form = new Form();
   form.name = 'Test Form';
   form.fields = [{ name: 'Field 1', type: 'text' }];
    await getConnection().getRepository(Form).save(form);
   // send a GET request to the API for the created form
   const response = await request(app).get(`/form/${form.id}`);
    // check that the response status is 200 OK
   expect(response.status).toBe(200);
   // check that the response body contains the expected form data
   expect(response.body.id).toBe(form.id);
   expect(response.body.name).toBe(form.name);
   expect(response.body.fields).toEqual(form.fields);
 });
 it('should return a 400 Bad Request error if the id is not a
number', async () \Rightarrow {
   // send a GET request to the API with a non-numeric id parameter
   const response = await request(app).get('/form/invalid');
   // check that the response status is 400 Bad Request
   expect(response.status).toBe(400);
    // check that the response body contains an error message
   expect(response.body.message).toBe('Invalid id parameter: must be
a number');
```

```
it('should return a 404 Not Found error if no form is found with the
given id', async () => {
    // send a GET request to the API with an id parameter for a non-
existent form
    const response = await request(app).get('/form/999');

    // check that the response status is 404 Not Found
    expect(response.status).toBe(404);

    // check that the response body contains an error message
    expect(response.body.message).toBe('Form not found');
});
});
```

This test spec uses the supertest library to make HTTP requests to the API and check the response status and body. The beforeAll, beforeEach, and afterAll hooks are used to set up and tear down the database connection and clear the database before each test. The three test cases check the different possible responses of the API: 200 OK for a valid id, 400 Bad Request for an invalid id, and 404 Not Found for a non-existent id.



Great i think we are done, thank you so much!



You're welcome! It was my pleasure to assist you. Don't hesitate to reach out if you have any more questions or need further help in the future.