A.2. Evaluation Instructions

Thanks for participating in our evaluation! Today, we want to test different approaches to generating OpenAPI specifications from Code. This means we would like you to implement and evolve the same application with 3 different controller layers. In order to get started quickly, we provide the application shell with a database, a data-access layer and required configuration in advance.

A.2.1. Initial survey

Before we get started, we'd like to know some general information about you:

- What's your current role?
- How many years of programming experience do you have?
- How familiar are you with TypeScript? (1-5)
- How familiar are you with nest's swagger capabilities? (1-5)
- How familiar are you with tsoa? (1-5)
- How familiar are you with swagger-jsdoc? (1-5)
- How familiar are you with express? (1-5)
- How familiar are you with the OpenAPI Specification? (1-5)

Along with these instructions, you will be provided with an order to complete this tasks in. We would kindly ask you to respect that order.

A.2.2. Getting familiar with the approaches

In case you are not familiar with NestJS:

NestJS[1] uses Controllers[2] to handle requests, which will return Data Transfer Objects (DTOs), which are classes with (public) properties. In order to document these Classes, NestJS uses property decorators [3]. In order to validate requests, usually validation decorators are required [4]. The Auto Validation pipe is already set up in the starter project.

[1] `https://github.com/nestjs/nest`
[2] `https://docs.nestjs.com/controllers`
[3] `https://docs.nestjs.com/recipes/swagger#decorators`
[4] `https://docs.nestjs.com/techniques/validation`

In case you are not familiar with tsoa:

Tsoa[5] compiles your code into an OpenAPI specifications and a small runtime layer at build time. This means tsoa can make use of TS interfaces and type definitions to generate documentation and validation logic. Additionally, tsoa uses JSDoc annotations to enhance documentation and validation[6] [7] [8].The starter project already handles serialization of Validation and Authorization Errors.

In both cases, methods with decorators and decorated parameters are used to inject parts of the request at runtime[9] [10]. Similarly, `@Request()` will inject the entire request but not produce documentation for access.

In case you are not familiar with swagger-inline/OpenAPI:

swagger-inline[11] is a small utility that allows you to write the OpenAPI specification side-by-side with your express code. If you do not feel comfortable writing OpenAPI by hand, we'd suggest you use a web UI[12].

Here's the list of starter projects:

- Approach 1: tsoa
- Approach 2: nest
- Approach 3: express + swagger-inline (oas)

For all 3 projects, the basic structure is already in place. Additionally, authentication is already implemented and ready to be used. All projects expose a Swagger UI on `http://localhost:3001/api`. As a point of reference, please take a look at the `UserController`.

The only requirement for running the projects is Docker (with docker-compose). To start, run `docker-compose up` inside the folder of each project.

A.2.3. Coding

In this step, we want to define a controller and data transfer objects for `Todos`. Each `Todo` belongs to a `User` who can create, retrieve, update, delete them.
Please note down the time needed to complete the objective for each task.

[5] `https://github.com/lukeautry/tsoa`
[6] `https://tsoa-community.github.io/docs/annotations.html`
[7] `https://tsoa-community.github.io/docs/descriptions.html#endpoint-descriptions`
[8] `https://tsoa-community.github.io/docs/examples.html`
[9] `https://docs.nestjs.com/controllers#request-object`
[10] `https://tsoa-community.github.io/docs/getting-started.html#defining-a-simple-controller`
[11] `https://github.com/readmeio/swagger-inline`
[12] `https://stoplight.io/p/studio`

Task 1: Implementing

First, we need to define some data transfer objects. In tsoa, we can use classes, but usually (annotated) TypeScript interfaces/type aliases will be sufficient.

The shape of the DTOs is based on the `TodoEntity`. Additionally, the requirements are:

- The title must have a `minLength` of 3.
- The description is returned as `null` if it wasn't set.

Todo DTO
The title, description and progress properties of the `TodoEntity`. (user is optional)

CreateTodo DTO

- A title of type string with a `minLength` of 3.
- An optional description of type string.
- A progress of type `TodoProgress`

UpdateTodo DTO

- Optional title of type string with a `minLength` of 3
- An optional description of type `string`.
- An optional progress of type `TodoProgress`.

All of the Todo Endpoints require an authorized `User`. Otherwise, the request should respond with a Status of 401 and a JSON Object with a message of type string. The User is defined on the request object and is provided:

- In tsoa by declaring `@Request() @InjectUser() user: User` as a parameter for the request handler
- In nest by using `@InjectUser() user: User`
- In express/swagger-inline as `request.user`

The Update and Delete Endpoint will be called with the UUID as a Path parameter and should respond with a Status of 404 and a JSON body with a message property of type string if the Todo to update is not present (or does not belongs to the user). All endpoints should respond with a Http response with status 400 and a JSON body with at least a message property of type `string`) (or `string[]`) if validation fails.

While the GET /todos endpoint responds with an array of Todos, the Create, Update and Delete Endpoints should respond with a single Todo Entity. Please implement the controller/-data transfer layer and document the API using OpenAPI.

Tips:

While there are a lot of similarities, all 3 approaches handle returning non-successful responses differently. While nest promotes throwing errors (with names based on the eventual status code) which require annotating the request handlers with `@ApiResponse` et al., which are caught and transformed to JSON responses, tsoa similarly uses
`@Response<T>(status, description, example)` to document responses as a result of error handling (Validation and Authorization Errors in middleware), but promotes injecting responders (`@Res() errorResponse: TsoaResponse<Status, Res, Headers>`) which can be called in a checked way instead. This is very similar to calling `res.status(400).json({})` in express directly.

Task 2: Improving

In this part, we will change the implementation of the GET /todos endpoint. We will introduce 2 optional query parameters, *progress* and *search*. The *progress* query param is of type *TodoProgress[]* and will be used to filter *Todos* by progress. The *search* of type string with a `minLength` of 3 can be used to search for a text. Add these 2 parameters to the endpoint, merge them into a *GetTodosDto* and pass it to the *getAll()* method of the `TodoService`.

Task 3: Modeling

In this part of the evaluation, we would like to explore modeling with all of these frameworks. Therefore, we created 2 requirements for Endpoints. The Documentation can be found here.

We have also provided stubs here.

Your task will be to add missing models and controllers.

The endpoint we use may belong to a rental company for boats and ships. There are multiple ships that can be rented (by passing the ship's id), or a boat, and while there are several boats, they are of the same type, so providing an id is not neccessary. To enable integration with other marketplaces beyond their homepage, the decision was made to publish this endpoint via API. The endpoint can be used by POSTing a request to /orders. The body should contain a JSON object with information about the order:

- a configuration, either for a ship or for a boat
- start date/time of the rental
- end date/time of the rental
- in case of a boat, the configuration must be an object which may contain an amount of lifevests, up to 8, which is the capacity of a boat
- in case of a ship, the configuration must contain the shipId annd a captainId, a reference to a captain the renter chose to accompany the trip. In case the renter is allowed to navigate the ship (will not be verified by the API), the capitainId should be explicitly set to null.
- a chargeAccountId, which references the account to charge for that purchase

Additionally, in order authorize the request, an Authorization header with a JWT must be provided.

There are several ways, the API will respond to these requests:

- 200: Ok: The rental was successful
- 400: The request failed because the rental could not be made (i.e. because a ship is not available that day). The response body will contain an object with a message explaining why the rental could not be processed.
- 401: Unauthorized: The Authorization header was not provided or incorrect
- 404: Not Found: One of the provided id's was not found. A message with details will be provided in the body
- 422: Unprocessable Entity: The request did not match the specification

A.2.4. Final survey

Thanks and congrats on completing these tasks. We would like you to answer a few questions about the approaches you got to know today.

- Would you prefer explicitly listing Parameters in Controllers (instead of grabbing them off the request object directly) in exchange for documentation and validation?
- Did you prefer to write descriptions/metadata in Decorator arguments or JSDoc?
- Did you enjoy using JSDoc as the source for descriptions?
- Did you enjoy using JSDoc for OpenAPI/JSON Schema modeling? Please provide reasons if you want to share any.
- Did you prefer classes over interfaces and type aliases to define DTOs?
- If your TS types were validated at runtime, would you still use class based DTOs? If the answer is yes, we would like to hear why.
- Which approach to writing the controller layer did you enjoy best overall?
- If requirements changed, which approach would be the least error-prone?
- Please share the time you needed for each task and framework.