**Exercise 6**

*Documenting your service with Swagger / OpenAPI*

**Prior Knowledge**

Basic understanding HTTP verbs, REST architecture

Service from Exercise 5

**Objectives**

Understanding Swagger and how to use tsoa to enable it in Typescript

**Software Requirements**

(see separate document for installation of these)

* Exercise 5 + requirements
* Swagger UI

**Overview**

The OpenAPI specification, which is still often known as Swagger - is an effective JSON/YAML model for describing RESTful services.

One of the main reasons we chose tsoa to build our service is that it takes an “OpenAPI-first” approach. This makes it very easy to modify our service to generate OpenAPI.

Steps

1. Start with your **purchase-starter** directory from the previous exercise.

(It should be passing all the tests.)

1. Before we do any coding, let’s look at the way the project is setup.  
     
   code tsoa.json  
     
   

Notice that this time there is an extra stanza in the file compared to the previous exercise. Lines 6-9 say to use OpenAPI spec version 3:  
 <https://swagger.io/specification/>   
and to output the swagger into the same directory as before (src/generated).

1. We can immediately generate the OpenAPI:  
     
   tsoa spec
2. This will create:  
     
    src/generated/swagger.json

Take a look. It will look a bit like:  


1. This is a valid Swagger/OpenAPI spec. However, we can improve it a bit.  
   You might question why I chose to generate it into the src folder?  
   We can get the service itself to serve this up, and even to serve a nice UI to visualise it, so it’s handy to consider it part of the src tree.   
     
   However, in a later exercise we are going to discover how we can use it in an API manager. In this model it would be better to export it somewhere else.
2. Let’s enable the “Swagger UI” to serve a nice webpage.  
     
   tsoa uses another package to do this - let’s install it:

yarn add swagger-ui-express  
 yarn add --dev @types/swagger-ui-express

1. Let’s add this to app.ts**:**In the imports add:  
     
   import \* as swaggerUi from "swagger-ui-express";

const swaggerDocument = import('./generated/swagger.json');

1. Then add (next to other app.use() code lines):  
     
   app.use("/api-docs", swaggerUi.serve,

async (\_req: express.Request, res: express.Response) => {

return res.send(

swaggerUi.generateHTML(await swaggerDocument)

);

});

1. What does this do? Basically when someone calls <http://localhost:8000/api-docs> this will generateHTML against the swagger doc that was generated by tsoa.
2. Before we run that, let’s edit the scripts in package.json to make sure that we always build the specs.  
     
   Change where it says “tsoa routes” to “tsoa spec-and-routes”  
   
3. Start your server: e.g.

yarn dev

1. Now browse to <http://localhost:8000/api-docs>



You should see a nice UI.  
  
Where did the name **purchase-tsoa** and version **1.0.0** come from?  
These are automatically picked up from the project name and version in package.json!

1. Try changing them. Notice that our nodemon watch doesn’t watch for package.json so you will have to restart your server.
2. Let’s try the service out from the Swagger UI.
3. Go to the button GET with no parameters:  
     
   Click on it.
4. Now click on Try it Out:  
   
5. You should see something like:  
   
6. Use this capability to GET and DELETE a specific uuid.
7. Now go to create a new order with POST:



1. This is pretty cool! The swagger generation has looked at the POCreationParams type (the subset of our purchase model), and the UI has now generated an acceptable JSON doc for you to edit.   
     
   Create a couple of orders this way.
2. This documentation is ok, but it is lacking a lot. Let’s go to the PurchaseController and document the @GET method.

Just above @Get(“/{uuid}”)   
  
type /\*\* and then hit Enter.  
  
vscode should automatically complete:



This is based on a system called JSDoc (<https://jsdoc.app/>)

Fill it in for example like this:



1. Note that the documentation for POST will probably not correctly describe the “successful” return code. We can fix that with a decoration of that method:  
     
    @SuccessResponse("201", "Created")   
    // Custom success response

@Post()

1. Assuming you have yarn dev and nodemon watching, now go and reload the Swagger UI:  
     
     
   Pretty sweet huh?
2. We can also decorate our model with jsdoc:  
   
3. This comes through in the schemas in the UI (see below the VERBS):  
   
4. Note that this is where our cool “Pick” approach would need tidying up a bit :-) The schemas that are generated aren’t that readable:
5. There is more you can do with tsoa to describe:

<https://tsoa-community.github.io/docs/descriptions.html>   
  
and to provide examples:  
<https://tsoa-community.github.io/docs/examples.html>

1. That’s the main lab concluded. There are some extensions if you want.

**Extension:**  
  
1. Take a look at redoc and redoc-cli

<https://github.com/Redocly/redoc>

<https://github.com/Redocly/redoc/blob/master/cli/README.md>

Use redoc cli to create a standalone HTML that documents your API

2. Sign up with Swagger Hub’s free trial:

<https://swagger.io/tools/swaggerhub/>

Use the tool to design an API from scratch. You can use your own ideas, but if you want one from me, how about creating an API for a simple todo list tracker.