**Exercise 5**

*Going through the maturity model*

**Prior Knowledge**

Basic understanding HTTP verbs, REST architecture

Some Typescript coding skill  
Exercise 3

A bit of Docker

**Objectives**

Build a fully functioning RESTful service with a database backend  
Understand the Richardson Maturity Model

**Software Requirements**

(see separate document for installation of these)

* Yarn, Node, Typescript, tsoa
* Postman
* VSCode

**Overview**

*The service we developed in Exercise 3 was not very exciting and did not implement a good RESTful design. It was designed to introduce Typescript, and understand the basics. Now we need to turn this into something useful.*

*You will need to write Typescript / tsoa code that adds RESTful decorations to create a service that meets a set of tests in Postman*

*There are a set of increasingly more demanding test cases that take us through Richardson’s REST maturity model.*

Your task is to make all those test cases pass.

**Introduction**

1. The instructions for this exercise are more freeform. By now you should have enough in your arsenal from Exercise 3 to be able to figure out the challenges.
2. I have created a project (like in Exercise 3). The project has all the Typescript boilerplate, etc

cd ~

git clone https://github.com/pzfreo/purchase-starter.git

cd purchase-starter

1. As usual, we need to install the dependencies in the package.json:

yarn install



1. It also includes a proper backend database using Postgres. You must have Postgres running. In this exercise I suggest you start a container to run Postgres, and then we will look at using docker-compose in a later exercise.  
     
   To start the Postgres database with the right setup, there is a simple script that uses docker:

./start-postgres.sh  
  
(All it does is this)  


1. You should see:  
   
2. The idea of this service is that we are going to allow users to post, put, get and delete PurchaseOrders from a system. The interface we want to design is a RESTful interface using JSON payloads. You will create orders by *posting* and/or *putting* JSONs. This will create a unique UUID and new hyperlink based on that UUID. You can then *get* or *delete* that resource using that hyperlink.
3. There is a **db.ts** that uses a Typescript package called typeorm to do Database access. This creates a repository (using the Singleton pattern) and then this is used by the service (as in Exercise 3).

You don’t need to understand typeorm. However, it is a nice utility so you might want to take a look:  
  
<https://github.com/typeorm/typeorm>   
  
Here is part of db.ts  


1. The really cool thing about TypeORM is that we can decorate our models:  
     
   

These decorations don’t interfere with tsoa using the model to create the front-end. So we end up with a nice codebase that doesn’t repeat models.

1. However, there may be aspects of the backend model that you don’t want to expose to the frontend. For example, we might want to create the date that a PO is created internally and not want that to appear in the body of the JSON that is sent to create it.  
     
   This is handled by using a cool Typescript feature which allows us to create a new type which is a subset of the last:  
    <https://www.typescriptlang.org/docs/handbook/utility-types.html#picktype-keys>

You can see this in the backend service here:



1. In order to make things easier for you, I have used the backend Service etc to create a WRONG BAD WONKY HTTP service. Do not treat this as a way to do things. Think of this as a non-RESTful, “Level 0” type service.

Why does this make it easier for you? Because all the code and syntax needed to implement HTTP services and to use the backend purchaseService is in the controller, so it should be clear how to add, remove, list and get entries in the backend database.

*Just look at* purchaseController.ts

1. What I’ve written can be summarised as RMM level 0: its using HTTP and JSON, but it is not RESTful. Your task is to convert it bit by bit into a RMM level 3 API.
2. You can run tests against it. These are in:   
     
    RMMlevel0.postman\_collection.json  
     
   As before you can run these with newman, import them into Postman UI, or there is a yarn script:  
     
   yarn run test-rmm0
3. The tests that you WANT to pass are in:  
     
    purchase-tests.postman\_collection.json  
     
   This defines the set of tests that you need the service to pass. There are 11 main tests and they follow the “Richardson Maturity Model”.  
     
   
4. The yarn scripts to run them are:

yarn run test-good

# (run once)

yarn run test-dev

#(run in watch mode)

1. Here is an overall summary of the behavior

|  |  |  |  |
| --- | --- | --- | --- |
| POST | / | Passes a representation of the order and create a new entry in the order database.  On success: HTTP 201 Created  Location header - URI of the new Resource The server’s representation of the resource is returned  Returns HTTP 400 Bad Request if bad JSON request sent | Consumes  application/json  Produces application/json |
| GET | /{id} | Get back a representation of order with identifier id.  If no such order is yet in the system, returns HTTP Not Found  If the order previously existed but has been deleted, returns HTTP Gone | Produces  application/json |
| PUT | /{id} | Updates an existing order  On success return HTTP 200 OK, together with the server’s representation of the updated order. If the JSON request is bad, return HTTP 400 Bad Request  If no such order is yet in the system, returns HTTP 404 Not Found  If the order previously existed but has been deleted, returns HTTP 410 Gone | Consumes  application/json Produces application/json |
| DELETE | /{id} | Marks an order as deleted  Returns HTTP 200 OK on success  If no such order is yet in the system, returns HTTP Not Found  If the order previously existed but has been deleted, returns HTTP 410 Gone | No body content |

1. Here is a brief description of each step you need to fix:

*Level 1 Basic URI and POSTing a JSON*

Let’s first support creating an Order.

The ideal behavior is that the POST would create a new resource (e.g. <http://localhost:8080/purchase/0123-456-789>) which was unique to this order. The return code should be 201 Created, and the POST body needs to contain the server’s representation of the resource. You also need the service to handle the error case correctly, which tsoa / models help with.   
  
Here are the two tests I consider roughly equivalent to L1.  
  


1. Level 2 - using the right HTTP Verbs with correct URIs, and good error codes (Get / Put / Delete)

These tests capture this:  


1. Level 3hypermedia - GET all ordersOur RESTful journey is nearly complete. In order to implement HATEOAS we need to support some links. A more developed HATEOAS application would offer links to other services/resources/APIs. For example, once the purchase order has shipped, we could include a link to the shipping tracker.   
     
   However, there is one simple resource we can offer, which is to provide a resource that “lists” the existing orders, using links.  
     
   This will return a JSON array of orders, in which each response is a valid link to the order. Here is a sample JSON output.

****Now get the final test to pass**

****

1. There is actually a bug in the poService.ts around GET all.   
   What is it?   
   Can you fix it?
2. That’s all. Congratulations.

**Extensions**

1. *Extension 1:*There is a serious problem with our GET all orders logic. What is it? How can we improve it?
2. *Extension 2:*

If you have completed all the above fully, you might wish to implement an improved flow.  
  
The proposed flow is that you allow an empty POST, which returns a location, followed by a PUT containing the order details, which “completes” the order.

*Why is this better than just a single POST?*