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

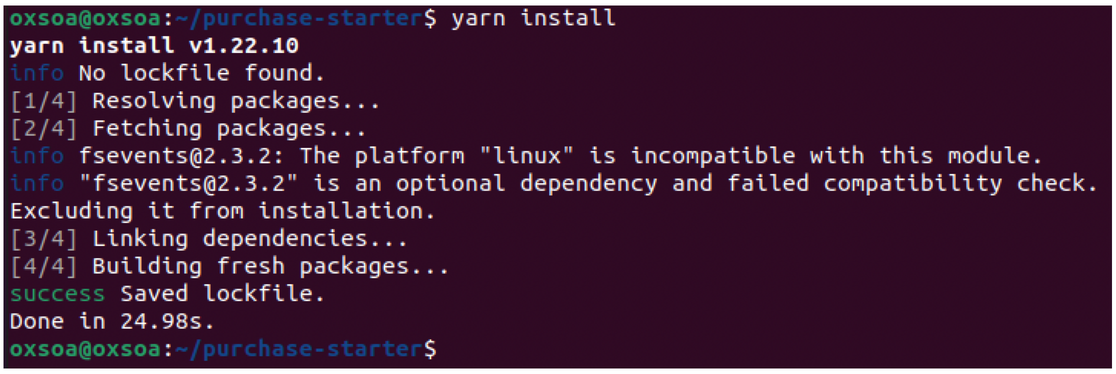
1. The instructions for this exercise are more freeform. By now you should have enough in your arsenal from Exercise 3 to be able to figure out the challenges.

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
```

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

```
yarn install
```



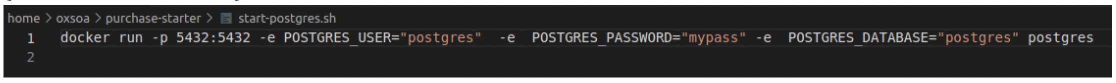
```
oxsoa@oxsoa:~/purchase-starter$ yarn install
yarn install v1.22.10
info No lockfile found.
[1/4] Resolving packages...
[2/4] Fetching packages...
info fsevents@2.3.2: The platform "linux" is incompatible with this module.
info "fsevents@2.3.2" is an optional dependency and failed compatibility check.
Excluding it from installation.
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Saved lockfile.
Done in 24.98s.
oxsoa@oxsoa:~/purchase-starter$
```

4. 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:

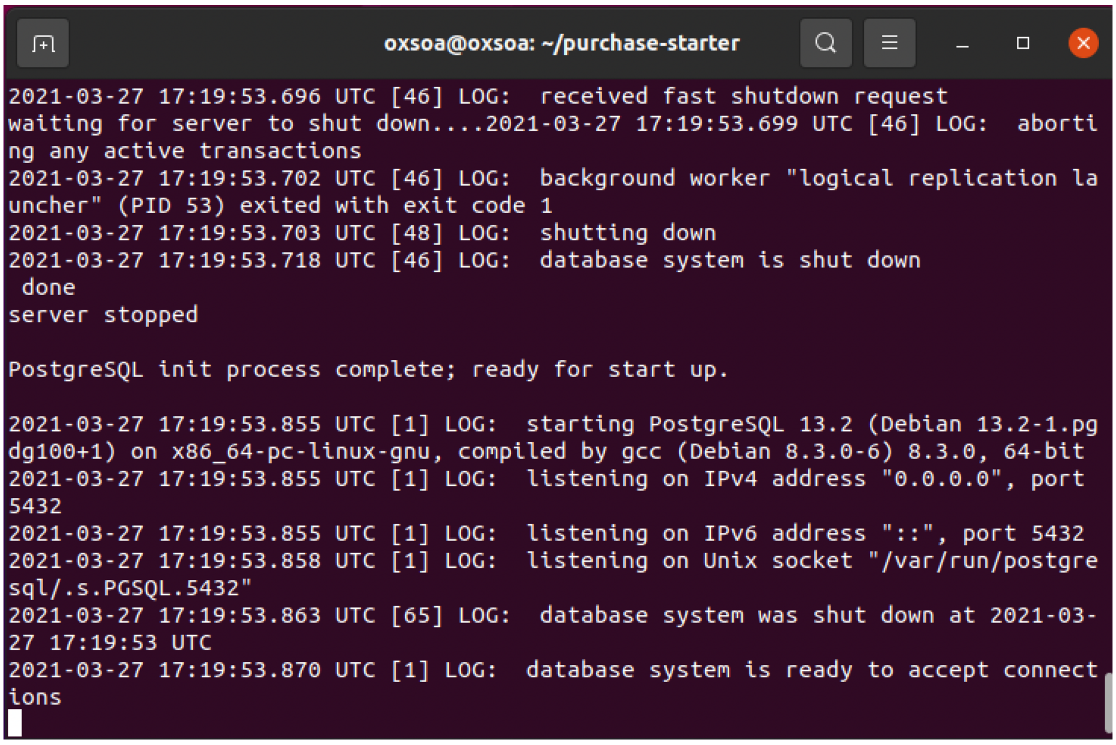
```
./start-postgres.sh
```

   (All it does is this)



```
docker run -p 5432:5432 -e POSTGRES_USER="postgres"  -e  POSTGRES_PASSWORD="mypass" -e  POSTGRES_DATABASE="postgres" postgres
```

5. You should see:



```
2021-03-27 17:19:53.696 UTC [46] LOG:  received fast shutdown request
waiting for server to shut down....2021-03-27 17:19:53.699 UTC [46] LOG:  aborti
ng any active transactions
2021-03-27 17:19:53.702 UTC [46] LOG:  background worker "logical replication la
uncher" (PID 53) exited with exit code 1
2021-03-27 17:19:53.703 UTC [48] LOG:  shutting down
2021-03-27 17:19:53.718 UTC [46] LOG:  database system is shut down
 done
server stopped

PostgreSQL init process complete; ready for start up.

2021-03-27 17:19:53.855 UTC [1] LOG:  starting PostgreSQL 13.2 (Debian 13.2-1.pg
dg100+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 8.3.0-6) 8.3.0, 64-bit
2021-03-27 17:19:53.855 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port
5432
2021-03-27 17:19:53.855 UTC [1] LOG:  listening on IPv6 address "::", port 5432
2021-03-27 17:19:53.858 UTC [1] LOG:  listening on Unix socket "/var/run/postgre
sql/.s.PGSQL.5432"
2021-03-27 17:19:53.863 UTC [65] LOG:  database system was shut down at 2021-03-
27 17:19:53 UTC
2021-03-27 17:19:53.870 UTC [1] LOG:  database system is ready to accept connect
ions
```

6. 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.

7. 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

```
import { createConnection, Repository } from "typeorm";
import { PurchaseOrder } from "./purchaseOrder";

let repositoryInstance : Repository<PurchaseOrder> | null = null;

export async function getRepository() : Promise<Repository<PurchaseOrder>> {
    if (repositoryInstance) {
        return repositoryInstance;
    }

    const conn = await createConnection({

        type: "postgres",
        host: process.env.DBHOST || "localhost",
        port: parseInt(process.env.DBPORT || "5432"),
        username: process.env.DBUSER || "postgres",
        password: process.env.DBPASSWORD || "mypass",
        database: process.env.DBDATABASE || "postgres",
        entities: [
            PurchaseOrder
        ],
        synchronize: true,
        logging: false

    });
    repositoryInstance =  conn.getRepository(PurchaseOrder);

    createSamplePO(repositoryInstance);

    return repositoryInstance;

}
```

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

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

@Entity()
export class PurchaseOrder  {

    @PrimaryGeneratedColumn("uuid")
    id: string;

    @Column({
        length: 256
    })
    poNumber: string;

    @Column("text")
    lineItem: string;

    @Column()
    quantity: number;

    @Column({ type: 'date' })
    date: Date = new Date();

    @Column({
        length: 256
    })
    customerNumber: string;

    @Column({
        length: 256
    })
    paymentReference: string;
```

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.

9. 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:

   ```
   // A post request should not contain an id or date
   export type POCreationParams = Pick<PurchaseOrder,
     "poNumber" | "lineItem" | "quantity" | "customerNumber" | "paymentReference">;
   ```

10. 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`

11. 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.

12. You can run tests against it. These are in:

    ```
    RMMlevel0.postman_collection.json
    ```

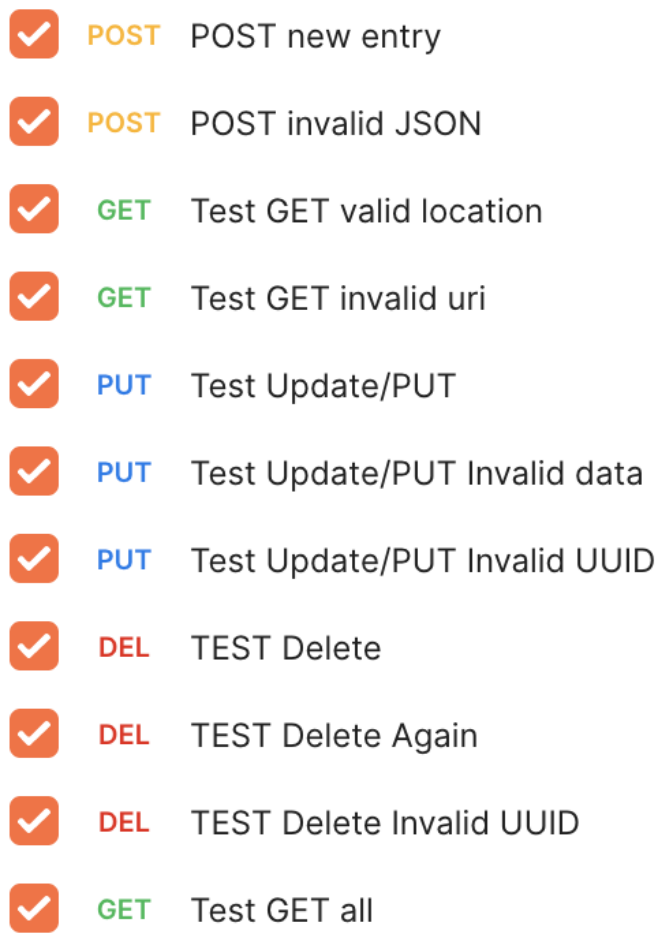
    As before you can run these with newman, import them into Postman UI, or there is a yarn script:

    ```
    yarn run test-rmm0
    ```

13. 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”.



✓	POST	POST new entry
✓	POST	POST invalid JSON
✓	GET	Test GET valid location
✓	GET	Test GET invalid uri
✓	PUT	Test Update/PUT
✓	PUT	Test Update/PUT Invalid data
✓	PUT	Test Update/PUT Invalid UUID
✓	DEL	TEST Delete
✓	DEL	TEST Delete Again
✓	DEL	TEST Delete Invalid UUID
✓	GET	Test GET all

14. The yarn scripts to run them are:

```
yarn run test-good
# (run once)

yarn run test-dev
#(run in watch mode)
```

15. 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

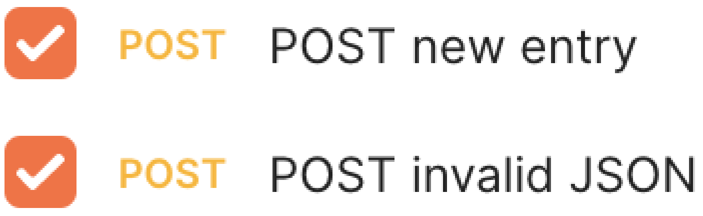
16. Here is a brief description of each step you need to fix:

    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.

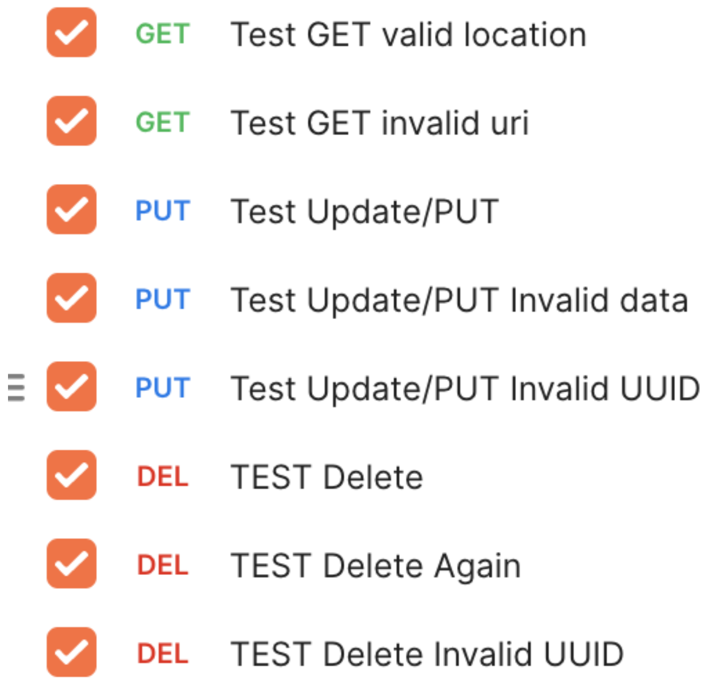


✓ POST POST new entry

✓ POST POST invalid JSON

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

    These tests capture this:



✓ GET Test GET valid location

✓ GET Test GET invalid uri

✓ PUT Test Update/PUT

✓ PUT Test Update/PUT Invalid data

✓ PUT Test Update/PUT Invalid UUID

✓ DEL TEST Delete

✓ DEL TEST Delete Again

✓ DEL TEST Delete Invalid UUID

18. Level 3 hypermedia - GET all orders

Our 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.

```
[
    {
        "href": "daedae31-aff3-49d0-91e5-d40062e192d4"
    },
    {
        "href": "9421f1d5-2ef3-455b-9d48-831f10e65432"
    },
    {
        "href": "5722704d-ca48-4081-a55d-7429cc5fa685"
    },
    {
        "href": "8ed0db5a-7f34-4a75-8269-1d96dfe3afcf"
    },
    {
        "href": "9b511d7d-7cc9-4bf8-b24f-e0e817d4c666"
    },
    {
        "href": "1fb5f795-b6e7-4503-94b0-a1ac3539957f"
    },
    {
        "href": "db058770-9e05-458f-a0ed-dcd02d586668"
    },
    {
        "href": "f672354b-bf89-40cf-939a-48f5f04fd597"
    }
]
```

Now get the final test to pass

✓ GET Test GET all

19. There is actually a bug in the poService.ts around GET all.
    What is it?
    Can you fix it?

20. 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?