318 Group 8

Project Task 2 Deliverable

Table of Contents

Project Structure

The Project layout for the Customer project has been adjusted since task 1, where the project layout adheres to the Doman Model pattern. Following the Domain-Driven Design building blocks for Domain Model Pattern; the customer, product and order projects have been adjusted to conform to this pattern.

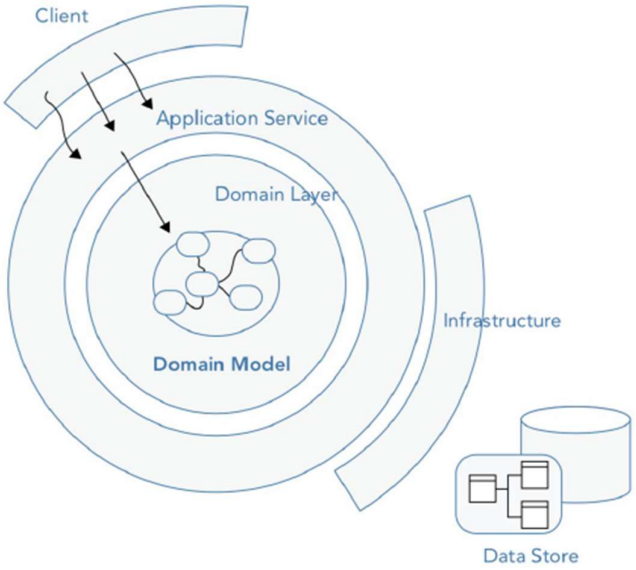


Figure 1.1. Domain Model pattern

UML diagrams are constructed to represent the Customer, Product and Order classes to represent the project to the Domain Model pattern accurately as possible.

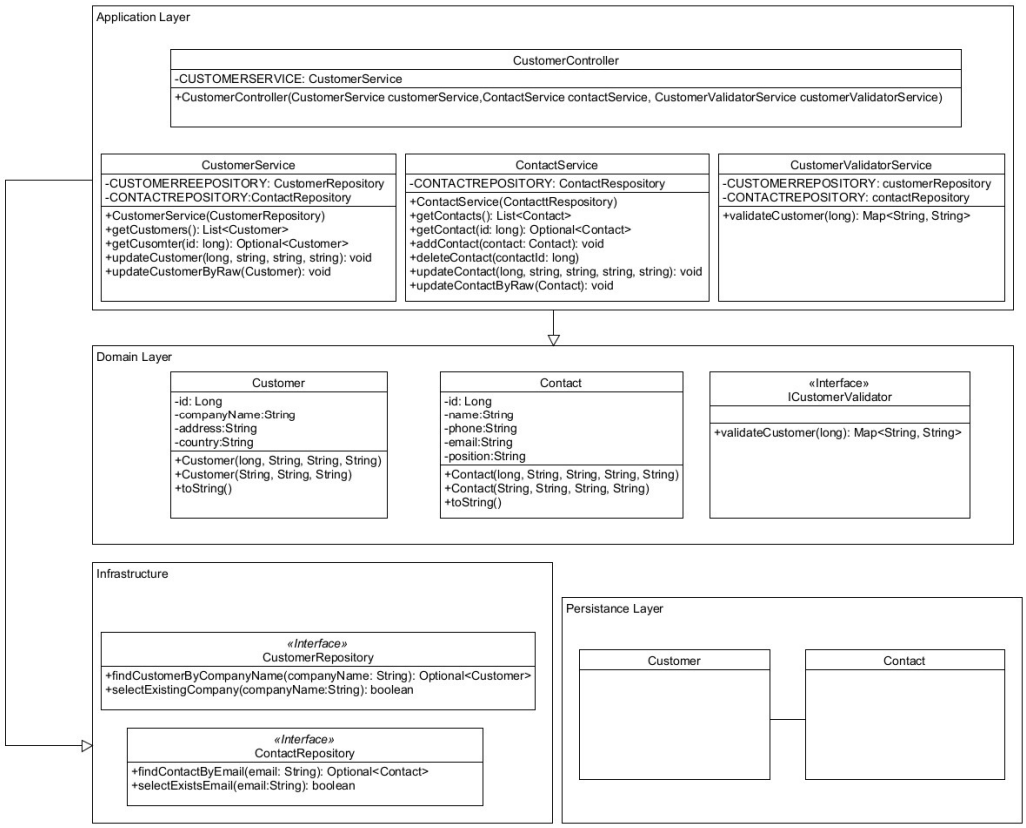


Figure 1.2 Customer Domain Model

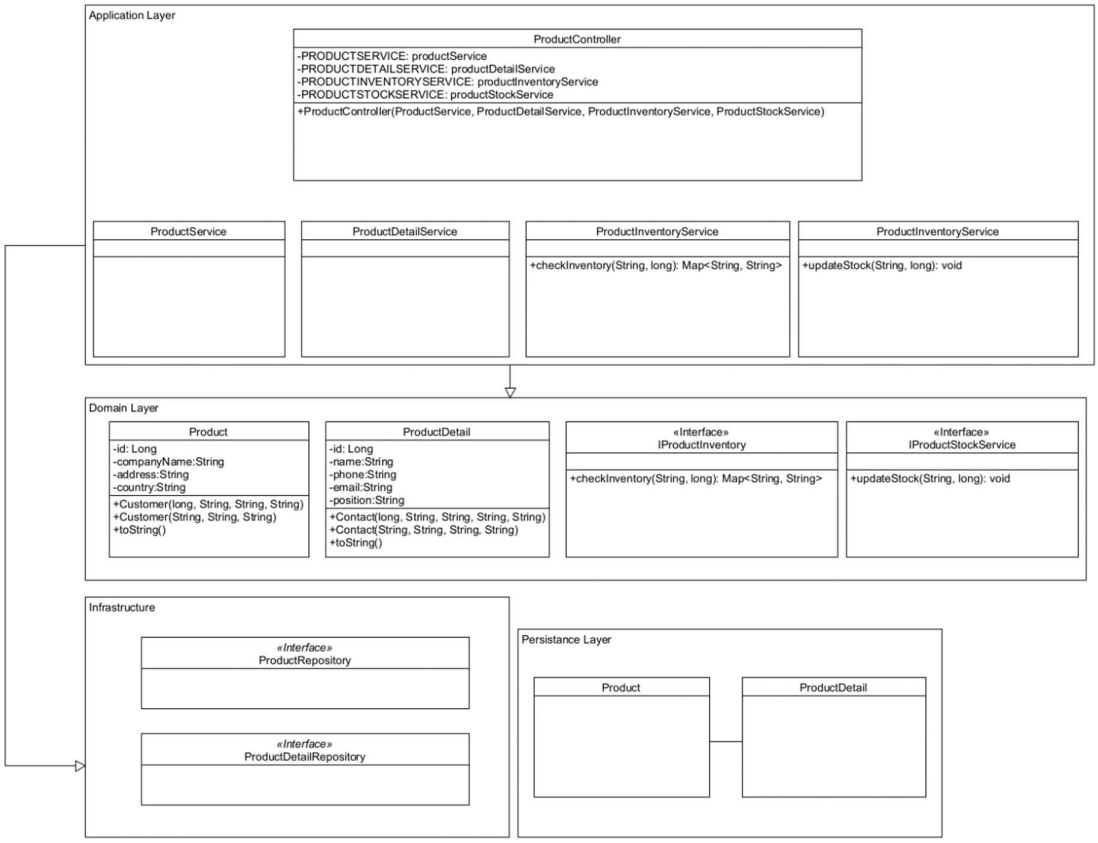


Figure 1.3 Product Domain Model

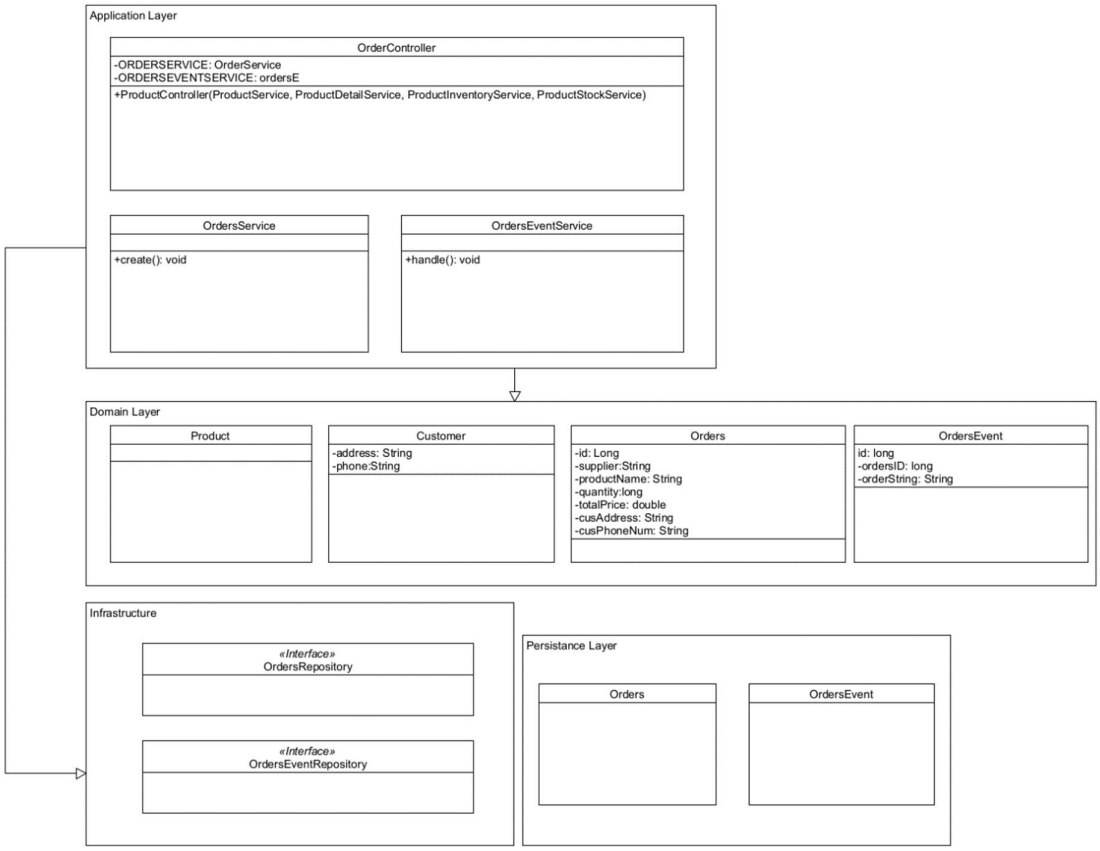
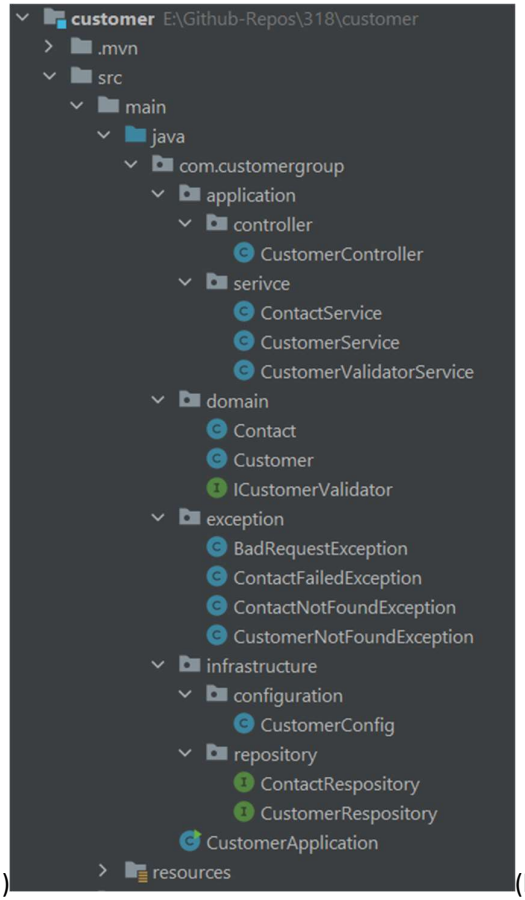


Figure 1.4 Order Domain Model

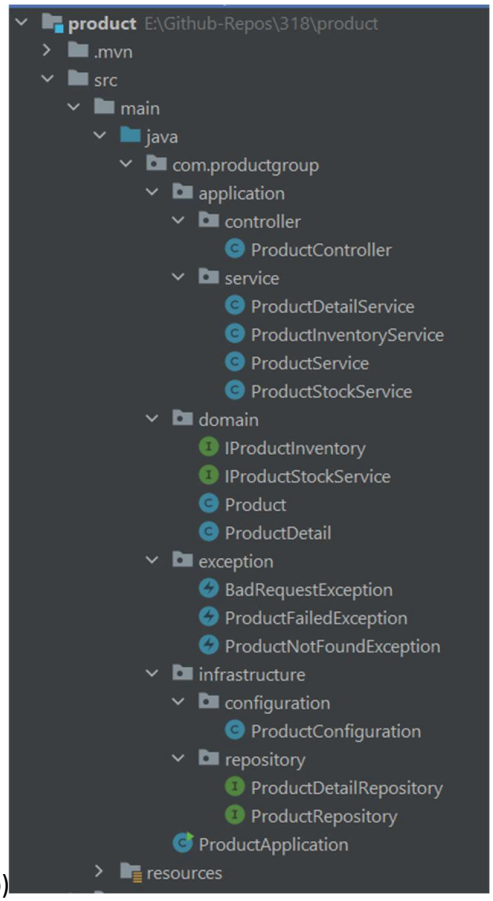
These UML diagrams replicate the Domain models as closely as possible in terms of the project structure. These models represent also models the N-tier architecture of each application. The representations do not include all functionalities but rather present how each tier is accessed and layered when the user interacts with the system. The project structures in the figure below replicate the N-tier architecture by layering the layers in packages. Except for the exception package, where this package acts as a layer that is not part of the N-tier architecture but rather a package for extending existing Exceptions for customized error messages.

When representing the N-tier architecture, the application layer contains the controller package or layer, which in turn acts as the API Layer. Where the Service Layer of the N-Tier architecture is represented by the Domain Layer. As for the data-access layer, this is contained in the infrastructure package with the persistence layer reflects off the domain models in a Table Module like pattern.

(a)


customer E:\Github-Repos\318\customer
.mvn
src
main
java
com.customergroup
application
controller
CustomerController
serivce
ContactService
CustomerService
CustomerValidatorService
domain
Contact
Customer
ICustomerValidator
exception
BadRequestException
ContactFailedException
ContactNotFoundException
CustomerNotFoundException
infrastructure
configuration
CustomerConfig
repository
ContactRespository
CustomerRespository
CustomerApplication
resources

(b)


product E:\Github-Repos\318\product
.mvn
src
main
java
com.productgroup
application
controller
ProductController
service
ProductDetailService
ProductInventoryService
ProductService
ProductStockService
domain
IProductInventory
IProductStockService
Product
ProductDetail
exception
BadRequestException
ProductFailedException
ProductNotFoundException
infrastructure
configuration
ProductConfiguration
repository
ProductDetailRepository
ProductRepository
ProductApplication
resources

(c)

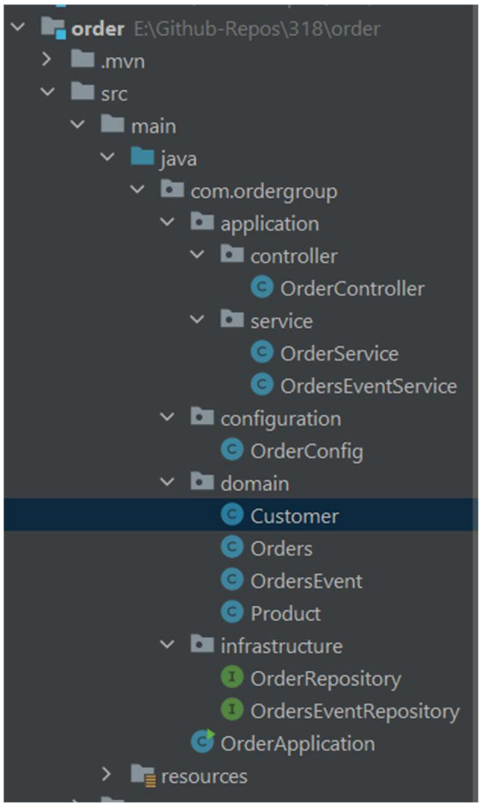

order E:\Github-Repos\318\order
.mvn
src
main
java
com.ordergroup
application
controller
OrderController
service
OrderService
OrdersEventService
configuration
OrderConfig
domain
Customer
Orders
OrdersEvent
Product
infrastructure
OrderRepository
OrdersEventRepository
OrderApplication
resources

Figure 2. Project Layout of (a) Customer, (b) Product and (c) Order applications.

As seen in figure 2, the packages have been aligned with the multi-tier architecture where the API layer containing the HTTP endpoints lies within the Application package. The 'Application' package contains both Application services and implemented Domain services for Customer, Product and Order as seen in figure 2(a), (b) and (c). This package also serves to orchestrate the REST services only (endpoints) and contain no business logic while domain services such as 'CustomerValidatorService', 'ProductInventoryService' and 'ProductStockStockService' contain business rules. The same applies to OrdersService. The domain services are implemented within the application package, by the implementation of the interfaces defined in the domain layer. Controllers for each application ***rest*** within the application package to maintain consistency of the Domain model, which is where the application layer maintains as an access endpoint for the user/client.

The domain layer/package contains the models of the domain and is separated from the infrastructure and the presentation/Application package. This is accessed by the application layer and by extension, is responsible for managing and persisting states for the infrastructure. Also, within the subdomain of the Customer domain, a one-to-one mapping exists between the Customer and Contact entities. The same applies for the Product domain, between the subdomains where a one-to-one mapping exists between the Product and ProductDetails classes.

The infrastructure package concerns itself with the Persistence layer, which maps the object models to the database models. With the Customer application, the Customer object is represented as a table in the repository/database and the Contact object is represented as a separated table within the same repository. This trend is the same for the Product application, where are Product and ProductDetail objects are represented as the Product and ProductDetail tables within another repository, separate from the Customer Application. As well as for the Order Application, where the Orders and OrdersEvent objects are represented as tables in a separate repository. This also enforces physical boundaries to retain the integrity and autonomous property of bounded contexts, known as shared-nothing architecture thus implements the strategic pattern of physical boundaries.

As seen in both the Customer and Product applications, there exists an exception package for the primary focus on extending exception classes to throw custom error messages. Also, these applications contain a configuration folder within the infrastructure package for "hard-coding" some data entries for development and testing purposes.

Functionality of each Java Class

Customer

Application	
CustomerController	API Layer - Handles HTTP Request Mapping
ContactService	Application Service - Handles request mapping specifically to Contact
CustomerService	Application Service - Handles request mapping specifically to Customer - Handles mapping requests to updating Customer and
CustomerValidatorService	Domain Service (Implementation) - Handles business logic - Demonstrates behaviour for validating the customer. Parse customerID for validation, if there are no contact details attached to the customer, throw an error.
Domain	
Contact	Entity - Identifies contact by: o Name o Phone o Email o Position o Assigned (Value object) Value Object - assigned: o Describes the properties of contact and the characteristics of the Contact entity being assigned to the Customer entity. o The sub-optimal design has been implemented where assigned has been set as long. o Described as a numerical value. o Behaviour-rich in the sense that it sets a flag when the Contact object has been assigned to Customer and checks if has not already been assigned to another customer.
Customer	Entity - Identifies Customers by: o companyName o address o country
ICustomerValidator	Domain Service - Contains an Interface for Domain Service (implemented by CustomerValidator in application>service layer)
Infrastructure	
ContactRespository	A repository for Contact with custom Queries
CustomerRespository	A repository for Customer with custom Queries
CustomerConfig	A configuration containing pre-set variables (to save time entering values since we're lazy developers).

Product

<table>
<tr><td colspan="2">Application</td></tr>
<tr><td>ProductController</td><td>API Layer
- Handles HTTP Request Mapping</td></tr>
<tr><td>ProductService</td><td>Application Service
- Handles request mapping specifically to Product entity and corresponding repository.</td></tr>
<tr><td>ProductDetailService</td><td>Application Service
- Handles request mapping specifically to ProductDetail and corresponding repository.</td></tr>
<tr><td>ProductInventoryService</td><td>Domain Service (Implementation)
- Handles business logic
- checkInventory() functionality checks if there’s enough quantity of the product. If there aren’t enough products, then this will throw an error.</td></tr>
<tr><td>ProductStockService</td><td>Domain Service (Implementation)
- Handles business logic
- updateStock() functionality updates the quantity of the product (only if the Order is successful).
-</td></tr>
<tr><td colspan="2">Domain</td></tr>
<tr><td>IProductInventory</td><td>Domain Service (Interface)
- An interface for the Domain Service for ProductInventoryService.
-</td></tr>
<tr><td>IProductStockService</td><td>Domain Service (Interface)
- An interface for the Domain Service for ProductStockService.</td></tr>
<tr><td>Product</td><td>Entity
- Identifies Product by:
o productCategory
o name
o price
o stockQuantity
o supplier
Value Object
- price
o Invariant check where if a price entered is low than 0.0, this will throw an error.
o Sub-optimal design has been implemented where representing the price as double has been chosen.
<pre>if(price < 0.0){
throw new ProductFailedException("You cannot enter a product with a negative price.");
}
this.price = price;</pre></td></tr>
</table>

	- stockQuantity
	  - Invariant check where if a stockQuantity entered is lower than 0, this will throw an error.
	  - Has been implemented as a long attribute rather than a class.
	`if(stockQuantity < 0){ throw new ProductFailedException("You cannot enter a product with negative quantity"); } this.stockQuantity = stockQuantity;`
	
	Contains a one-to-one mapping with Product Detail ID

ProductDetail	Entity
	- Identifies ProductDetail by:
	  - Description
	  - Comment
	Contains a one-to-one mapping with product based on ID

IProductInventory	Domain Service - Contains an Interface for Domain Service (implemented by ProductInventoryService in application>service layer)
IProductStockService	Domain Service - Contains an Interface for Domain Service (implemented by ProductStockService in application>service layer)
Infrastructure	
ProductDetailRepository	A repository for ProductDetail with custom Queries
ProductRespository	A repository for Product with custom Queries
ProductConfiguration	A configuration containing pre-set variables (to save time entering values since we're lazy developers).

Order

Application	
OrderController	API Layer - Handles HTTP Request Mapping
OrderService	Application Service - Handles request mapping specifically to Orders Entity as well corresponding repository - Further explanations of how OrderService meets aggregates will be explained in the next section.
OrdersEventService	Application Service - Implements event listener to handle OrdersEvent events and corresponding repository.
Domain	

Orders	Entity
	- Identifies Orders by:
	  - supplier
	  - productName

	○ quantity ○ totalPrice ○ CusAddress ○ cusPhoneNum
OrdersEvent	Entity - Identifies OrdersEvent by: ○ OrderID ○ orderString
Customer	Entity - Identifies Customer by: ○ address ○ phone
Product	Entity - Identifies Product by: ○ productName ○ price ○ supplier
Infrastructure	
OrderRepository	A repository for Orders
OrdersEventRepository	A repository for OrdersEvent

Tactical Pattern – Domain Service

Domain Services have been implemented in the Customer application where ICustomerValidator represents the interface for the domain service. This interface contains the function validateCustomer, which enforces the business logic that *"a customer is only valid if they have contact details*". The implementation of this logic exists in the service layer where the CustomerValidatorService implements this class.

As for the Production application, interface IProductStockService represents the business logic for updating stock once an order has been made and IProductInventory represents the business logic for checking the inventory if there's enough quantity of the product. These interfaces exist in the domain package, where the implementations exist in the application service layer as seen in the ProductStockService and ProductInventoryService classes.

Specific to the Order application, domain events have also been implemented where the OrderService consist of the use of the ApplicationEventPublisher library as seen in the code below.

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private RestTemplate restTemplate;

    private ApplicationEventPublisher publisher;

    @Autowired
```

```
    public OrderService(OrderRepository
orderRepository, ApplicationEventPublisher publisher,
RestTemplateBuilder builder) {
        this.orderRepository = orderRepository;
        this.publisher = publisher;
        this.restTemplate = builder.build();
    }
```

This service publishes an event to the EventListener bean that calls for the handle() function that exists OrdersEventService class as seen in the code below.

```
@EventListener
public void handle(OrdersEvent ordersEvent){
    try {
        ordersEventRepository.save(ordersEvent);

        //need to update stock
        System.out.println("Received order event, no
cap");
        String getOrderURL =
"http://localhost:8082/api/orders/" +
ordersEvent.getOrderID();
        System.out.println(getOrderURL);

        Orders order =
restTemplate.getForObject(getOrderURL, Orders.class);

        String updateStockUrl =
"http://localhost:8081/product/" +
order.getProductName() + "/quantity/" +
order.getQuantity();
        System.out.println(updateStockUrl);
        restTemplate.exchange(updateStockUrl,
HttpMethod.PUT, new HttpEntity<>(order, new
HttpHeaders()), Void.class);
    } catch(Exception e) {
        System.out.println("Message: " +
e.getMessage());
        throw new ResponseStatusException(
                HttpStatus.INTERNAL_SERVER_ERROR,
e.getMessage(), e
        );
    }
}
```

Tactical Pattern – Value Objects

Value Objects can be seen earlier within the Product class where price and stock quantity exist as integrated primitive data types in the class. This implement can be considered as a sub-optimal design choice due to the high coupling and cohesion between the value objects and the domain concept. However, reasonable assumptions that price and stock quantity cannot be negative justifies this sub-optimal design as it can be assumed (as of this current time in the project) that there is no more self-validation is needed.

Tactical Pattern - Aggregates

Within OrderService contains a create function that follows the Tactical pattern of Aggregates by preferring IDs over object references where an order is created by the use if the customer's ID as opposed to the customer object.

```
public void create(long custID, String productName,
long quanitity){

    try {
        // validate customer
        String validateURL =
"http://localhost:8080/api/customer/validate=" +
custID;
        Customer customer =
restTemplate.getForObject(validateURL, Customer.class);

        // check product inventory
        String checkInvURL =
"http://localhost:8081/product/checkInventory/productNa
me=" + productName + "/quantity=" + quanitity;
        Product product =
restTemplate.getForObject(checkInvURL, Product.class);

        double totalPrice =
(Double.parseDouble(product.getPrice()) * quanitity);
        // order Event
        //create an order
        Orders order = new Orders(product.getSupplier()
                , productName
                , quanitity
                , totalPrice
                , customer.getAddress()
                , customer.getPhone()
        );

        orderRepository.save(order);
        //order Event
        OrdersEvent ordersEvent = new
```

```
OrdersEvent(order);
        publisher.publishEvent(ordersEvent);
    }
    catch (Exception e){
        System.out.println("Message: " +
e.getMessage());
        throw new ResponseStatusException(
                HttpStatus.INTERNAL_SERVER_ERROR,
e.getMessage(), e
        );
    }
}
```

Examples of Input and Output

Original GET request to customer

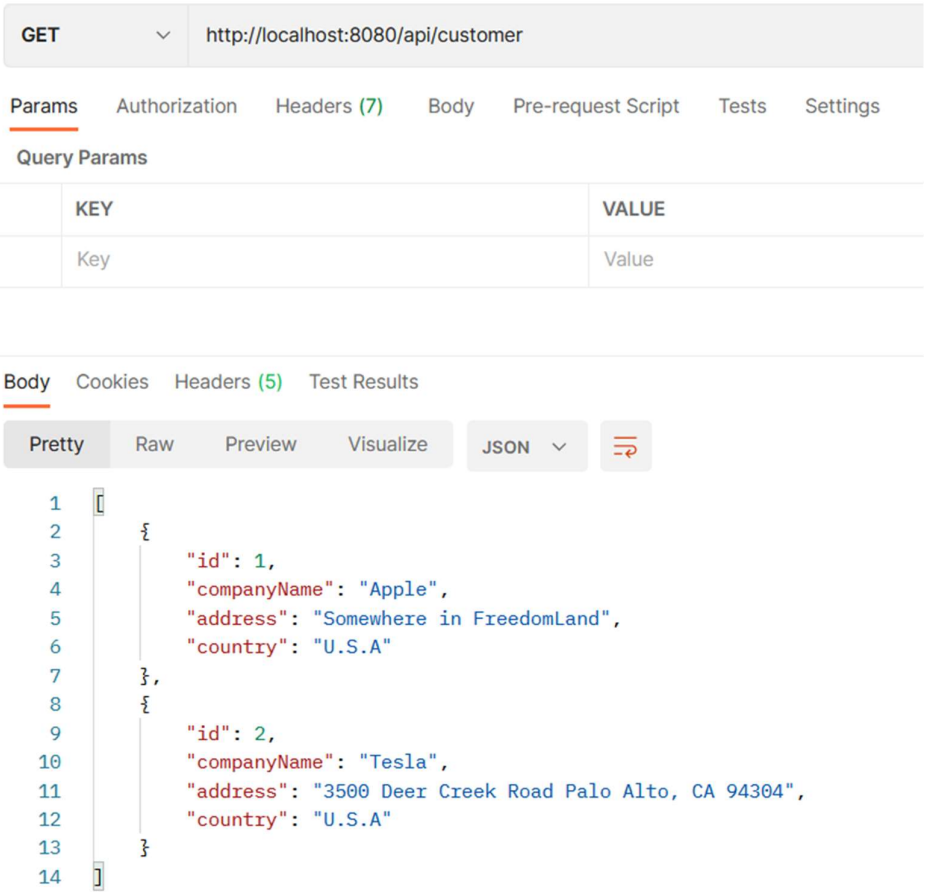


Figure 3.1 – GET request for customer

USE CASE 1: Create/update customer

Create: POST request to customer

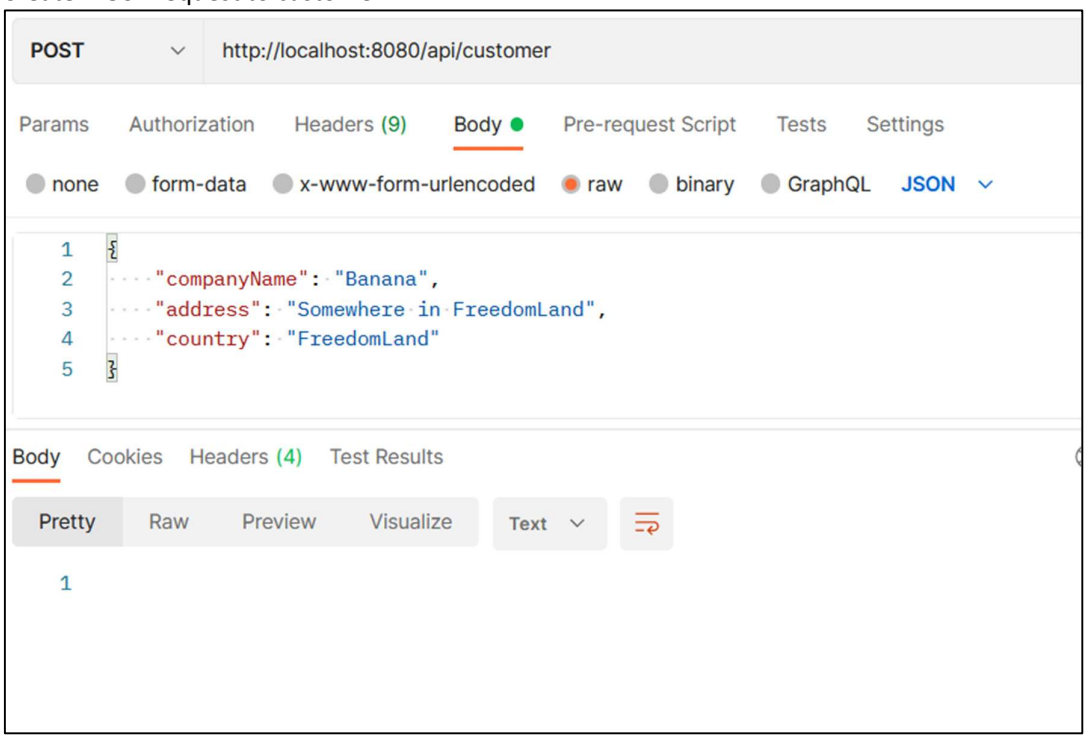


Figure 4.1 – POST request for customer

To create a customer, a POST request is sent to api/customer endpoint with a body of raw text in JSON. An example of this in curl is provided below.

Curl example:

```
curl --location --request POST 'http://localhost:8080/api/customer' \
--header 'Content-Type: application/json' \
--data-raw '{
    "companyName": "Banana",
    "address": "Somewhere in FreedomLand",
    "country": "FreedomLand"
}'
```

Note: The value of each key can be adjusted if needed, in JSON format.

GET customers after POST request

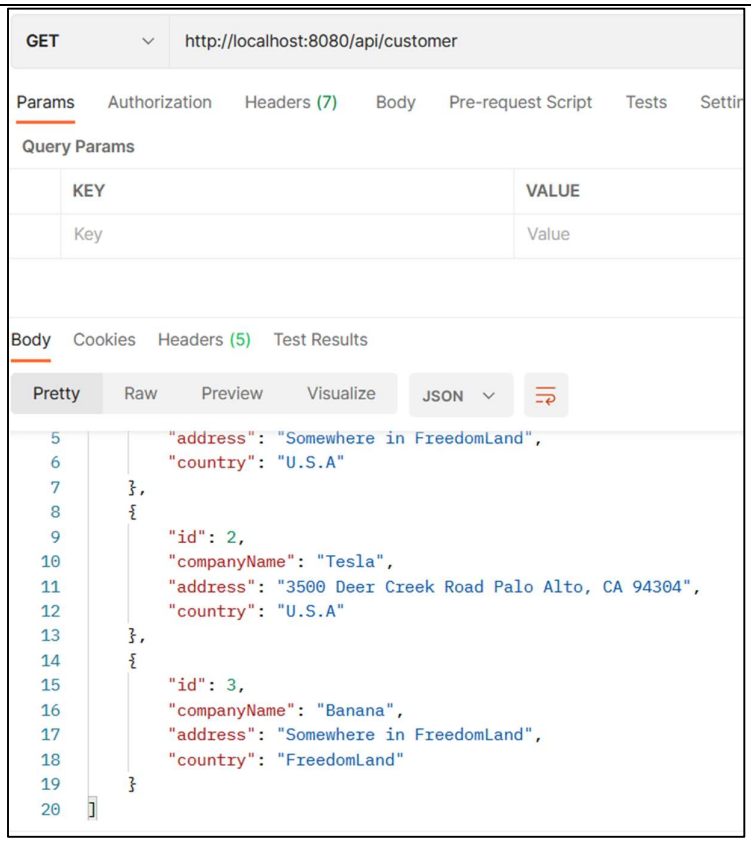


Figure 4.2 – GET Request after successful 'Create Customer'

In Figure 4.2 – it can be seen that the customer has been successfully inserted into the repository. A GET request to the endpoint 'api/customer' displays all the objects created in the Customer Repository with the new addition of the **Banana** company created in previously.

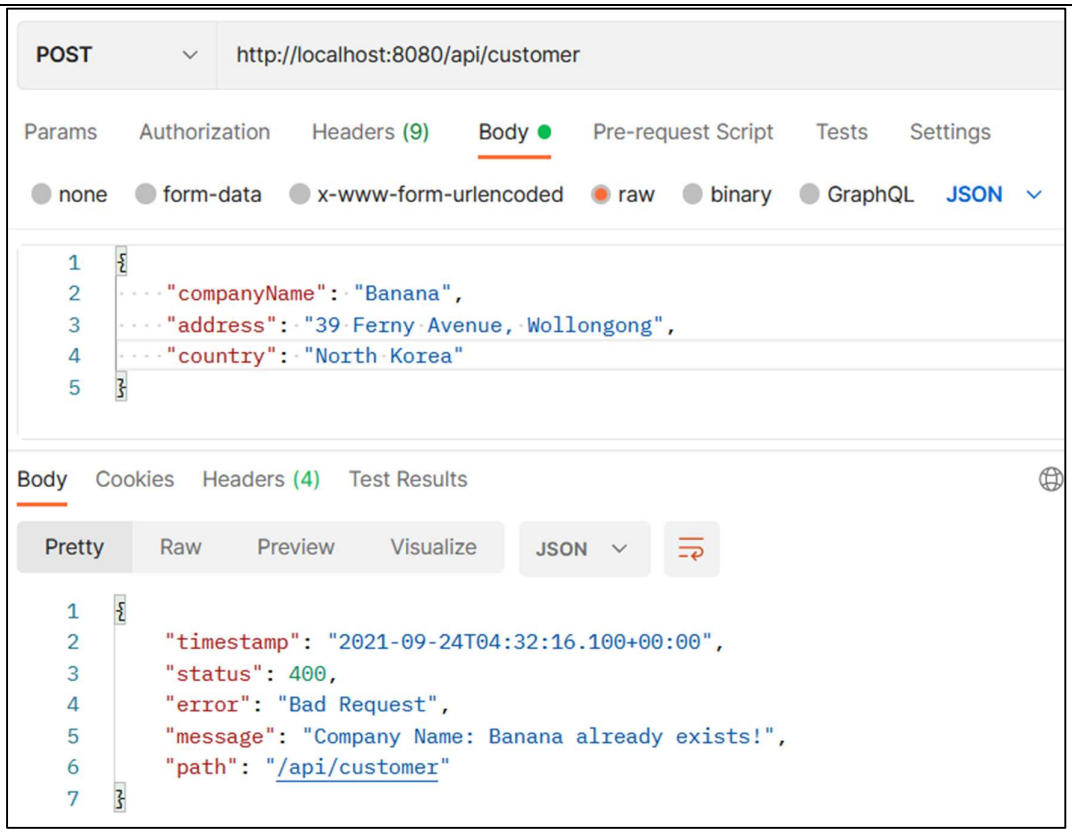


Figure 4.3 – POST Request containing same companyName

In entering a new customer with a companyName that already exists within the repository will throw an error as seen in figure 4.3.

UPDATE CUSTOMER

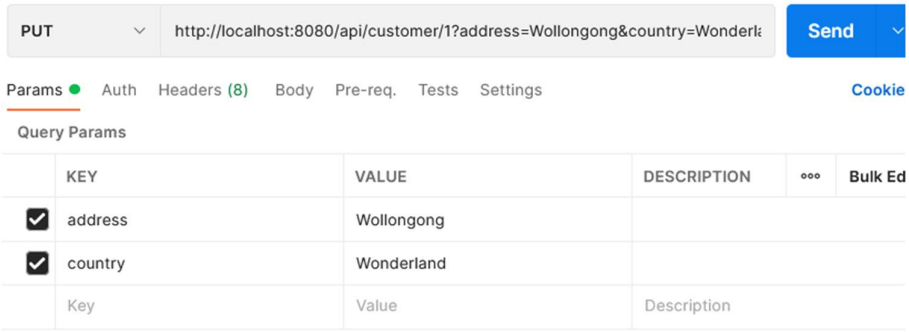


Figure 4.4 – PUT Request to customer with ID = 1

A PUT request can be sent to the customer where id is 3. Key parameters can be used to update the companyName as seen in figure 4.4 where the key companyName is set to the value “Banana”.

```
curl --location --
request PUT 'http://localhost:8080/api/customer/3?companyName=Banana'
```

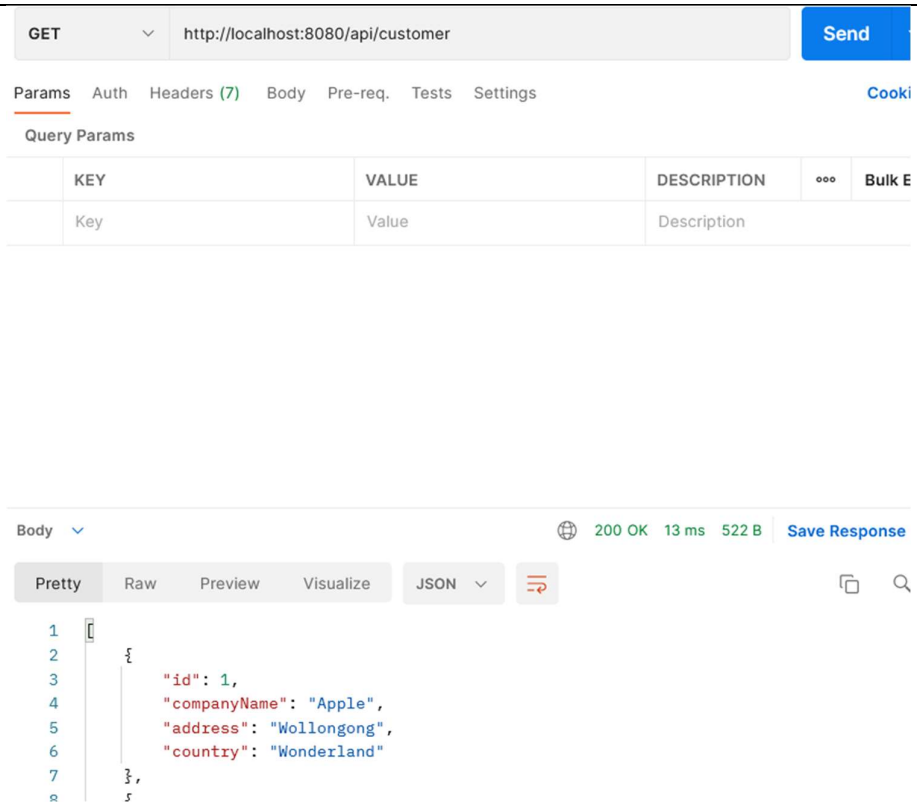


Figure 4.5 – GET Request after successful update

PUT request failed because companyName “Tesla” already exists

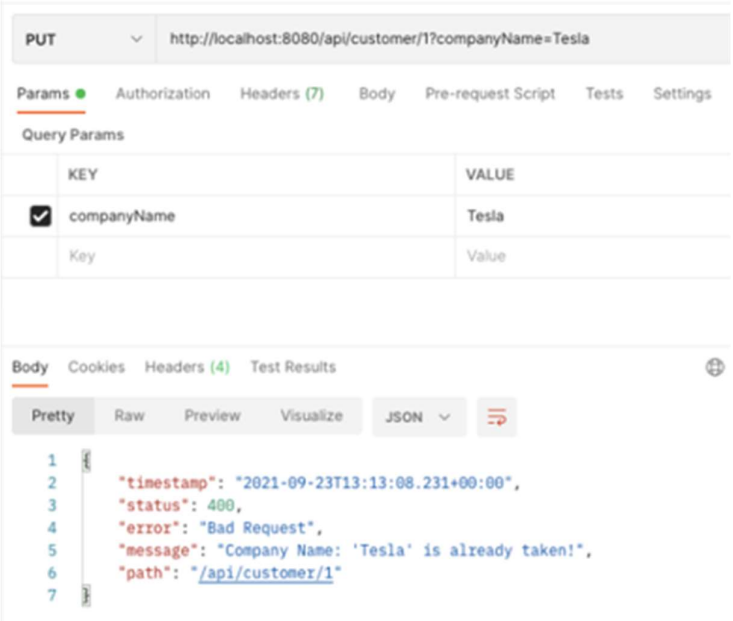


Figure 4.6 – Updating customer with a company name “Tesla” which already exists in the database.

USE CASE 2: Create/update customer contact

Create Contact
Input:

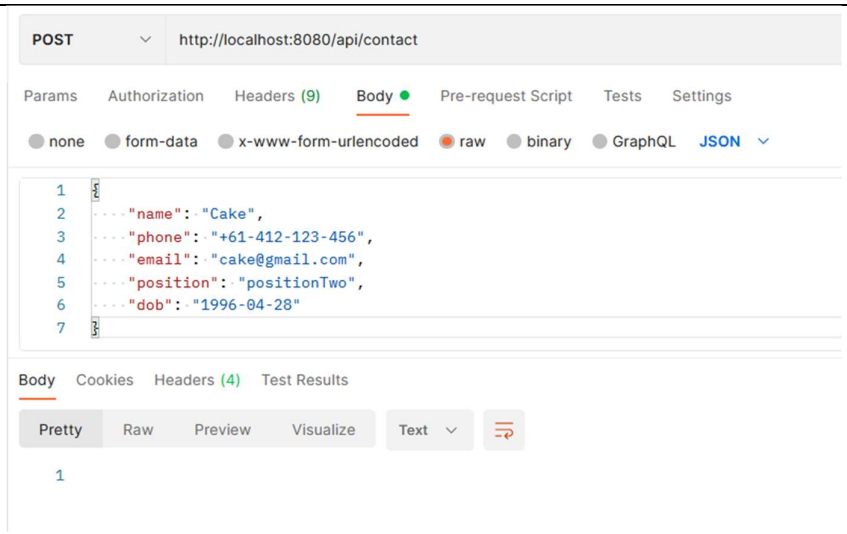


Figure 5.1 – Updating customer with a company name “Tesla” which already exists in the database.

When sending a POST request to the contact API of creating a contact, a successful creation will result in no body. One value to keep in mind is the name attribute; where if there is a name that already exists in the database, this will throw an error as an assumption is made that each name is unique.

Curl:

```
curl --location --request POST 'http://localhost:8080/api/contact' \
--header 'Content-Type: application/json' \
--data-raw '{
    "name": "Cake",
    "phone": "+61-412-123-456",
    "email": "cake@gmail.com",
    "position": "positionTwo",
    "dob": "1996-04-28"
}'
```

Note: The value of each key can be adjusted if needed, in JSON format.

Output:

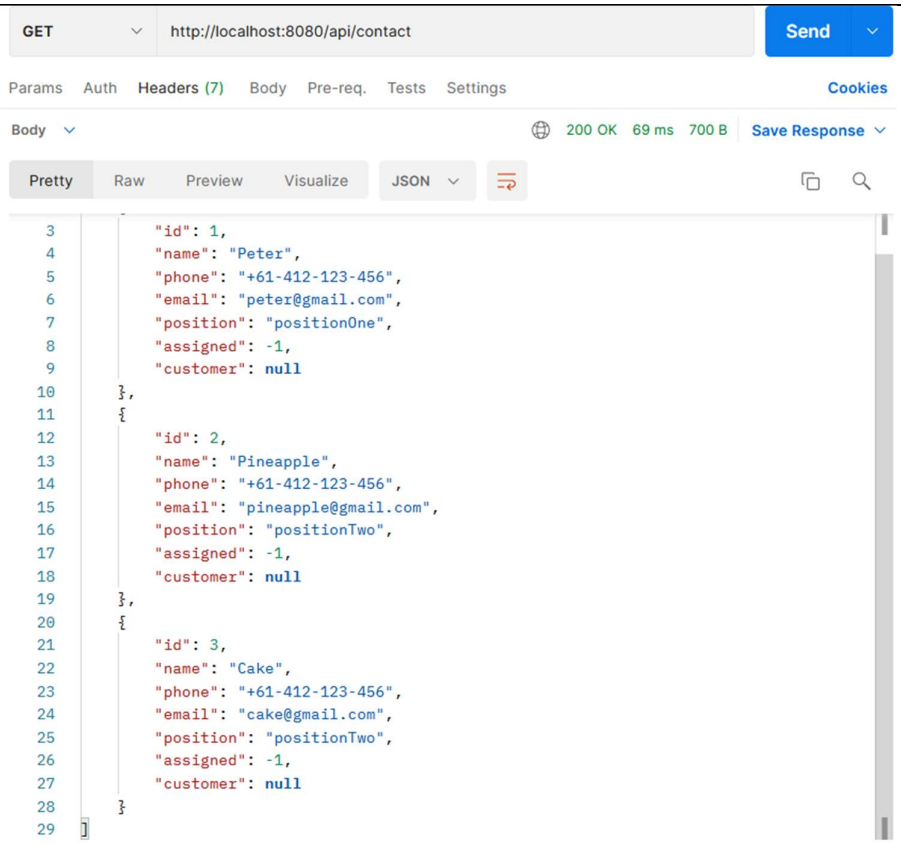


Figure 5.2– Output of creating a new contact

After successful POST, we can view the new entry with a GET request to the contact endpoint as seen in figure 3.9.

Curl:

```
curl --location --request GET 'http://localhost:8080/api/contact'
```

Update Contact

Input:

PUT request can be sent to api/contact/{contactId} using key parameters where the key value are attributes of Contact and values containing the value to update/assign the contact object.

Example:

http://localhost:8080/api/contact/3?name=TheApple&phone=911&email=apple@apple.com&position=positionfour

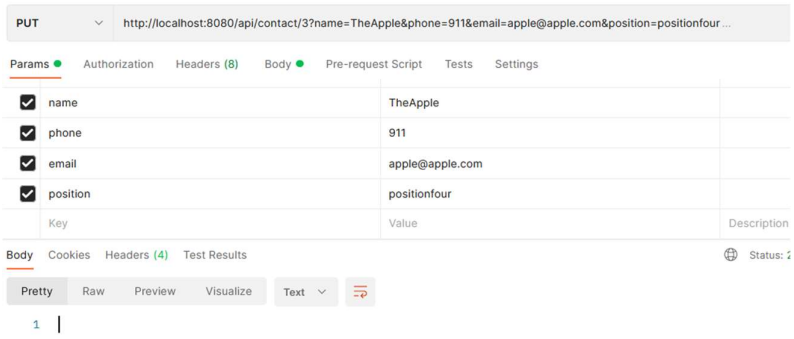


Figure 5.3 – Update request of updating a contact.

Curl equivalent:

```
curl --location --
request PUT 'http://localhost:8080/api/contact/3?name=TheApple&phone=911&email=apple@apple.com&position=positionfour'
```

Output:

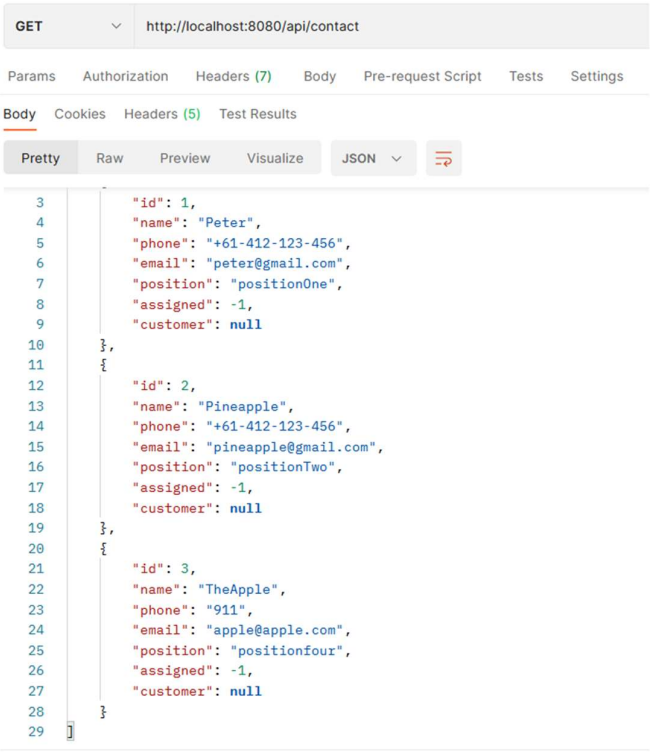


Figure 5.4 – Output of updating contact 3

Update Customer with contact details

Customers can be updated with adding contact details by sending a PUT request to Api/customer/{customerId}/contact/{contactId}

Where {customerId} and {contactId} are both replaced by a number corresponding to the ID of the objects.

Input:

```
curl --location --request PUT 'http://localhost:8080/api/customer/2/contact/1'
```

This is where customer with the ID 2 is assigned with the contact of ID 1.

Output:

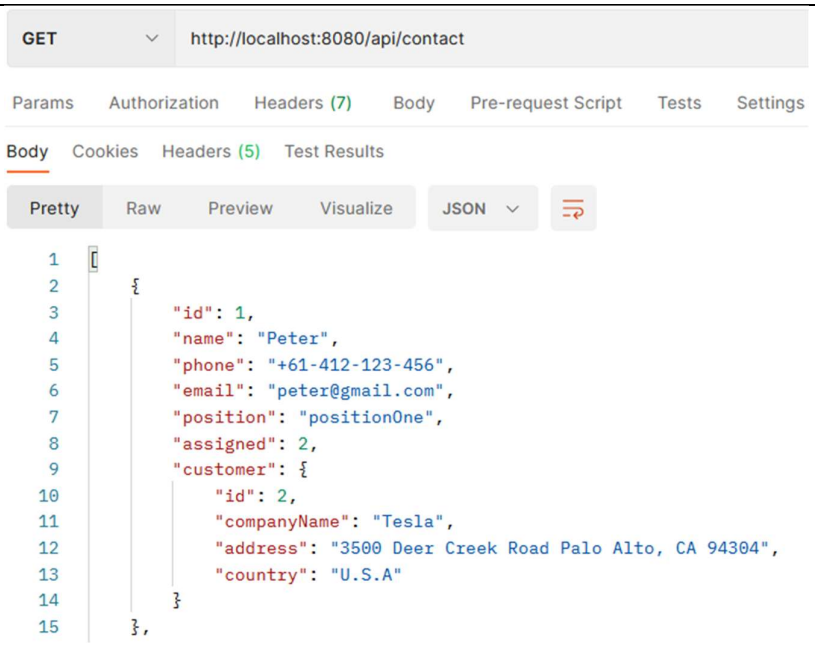


Figure 5.5 – Output of updating contact 3

USE CASE 3: Look up customer basic info and contact

Look up customer basic info can be achieved via the contact API, this is due the JSON ignore defined in the Customer.java entity class.

To grab all customer basic info and contact, a GET request can be sent to 'api/contact'.

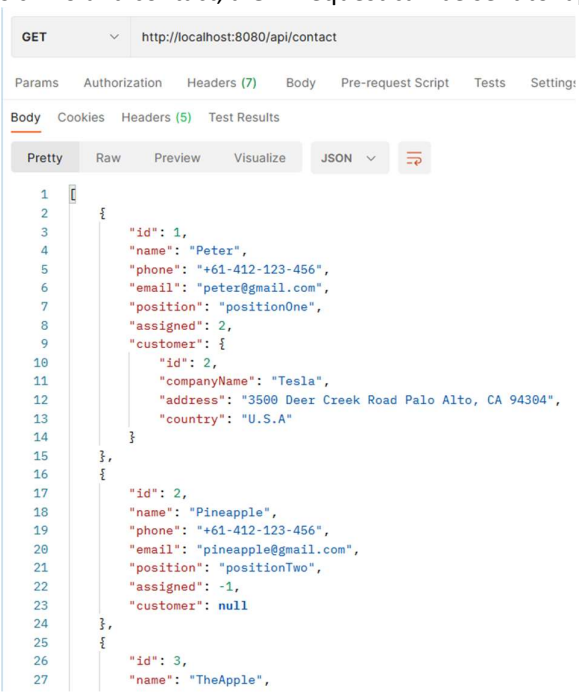


Figure 6.1 – Looking up all customer basic info and contact

or

```
curl --location --request GET 'http://localhost:8080/api/contact'
```

Looking up a specific customer basic info and contact

This can be achieved by sending a GET request to the contact endpoint with an ID corresponding to the contact ID. For example,

Input:
http://localhost:8080/api/contact/1
or

```
curl --location --request GET 'http://localhost:8080/api/contact/1'
```

Output:

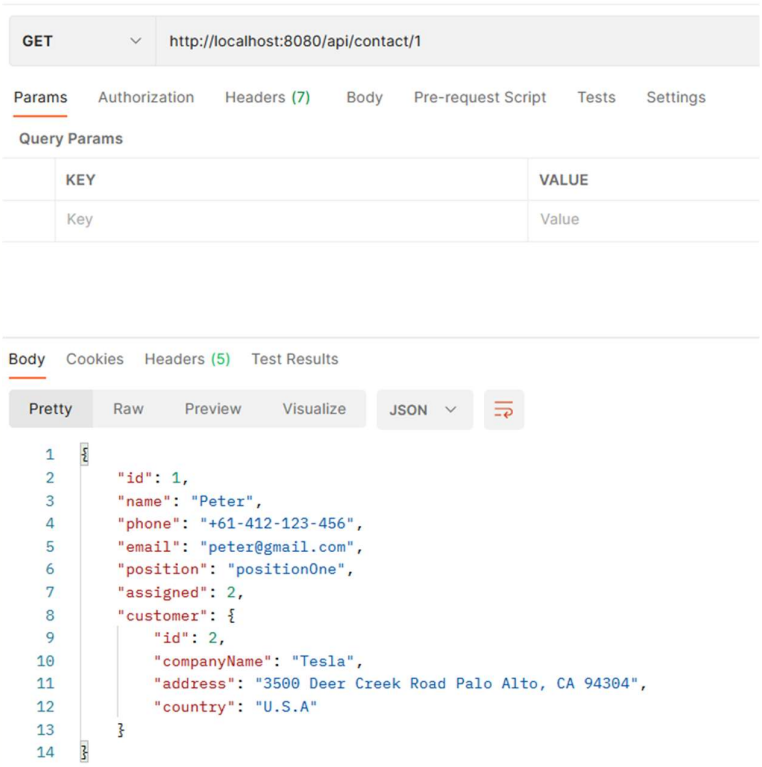


Figure 6.2 – Looking a specific contact detail.

Looking up a contactID allows us to view the contact details and the customer it has been assigned. If there were to be no customer assigned to the contact details, this will print:

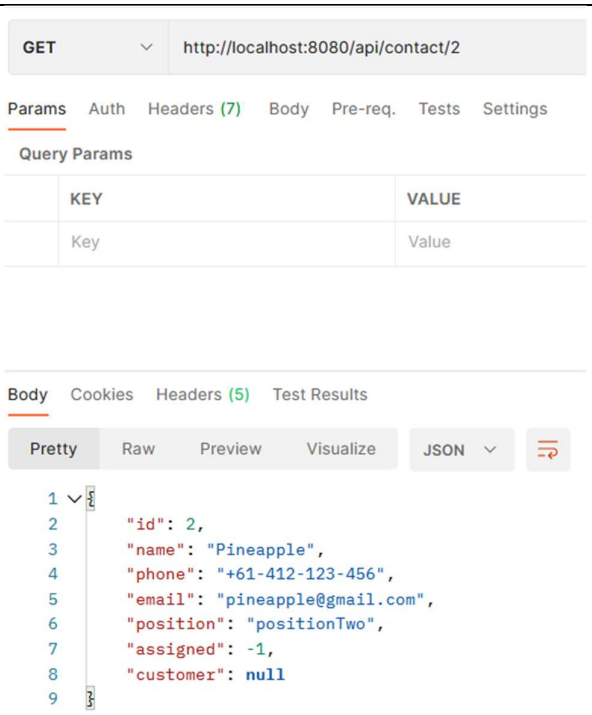


Figure 6.3– Looking a specific contact containing no assignment to Customer.

USE CASE 4: Create/update product

Create: POST request to product

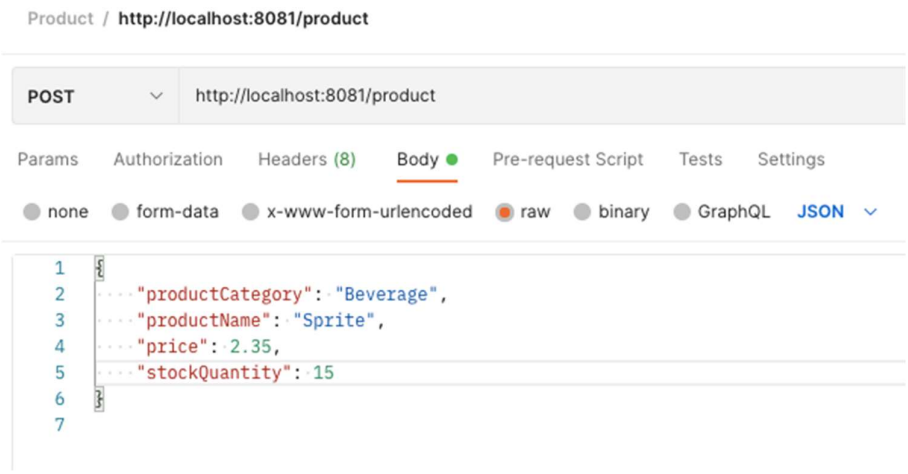


Figure 7.1 – POST request for product

To create a product, a POST request is sent to /product endpoint with a body of raw text in JSON. An example of this in curl is provided below.

Curl example:

```
curl --location --request POST 'http://localhost:8081/product' \
--header 'Content-Type: application/json' \
```

```
--data-raw '{
    "productCategory": "Beverage",
    "productName": "Sprite",
    "price": 2.35,
    "stockQuantity": 15
}
'
```

Note: The value of each key can be adjusted if needed, in JSON format.

GET products after POST request

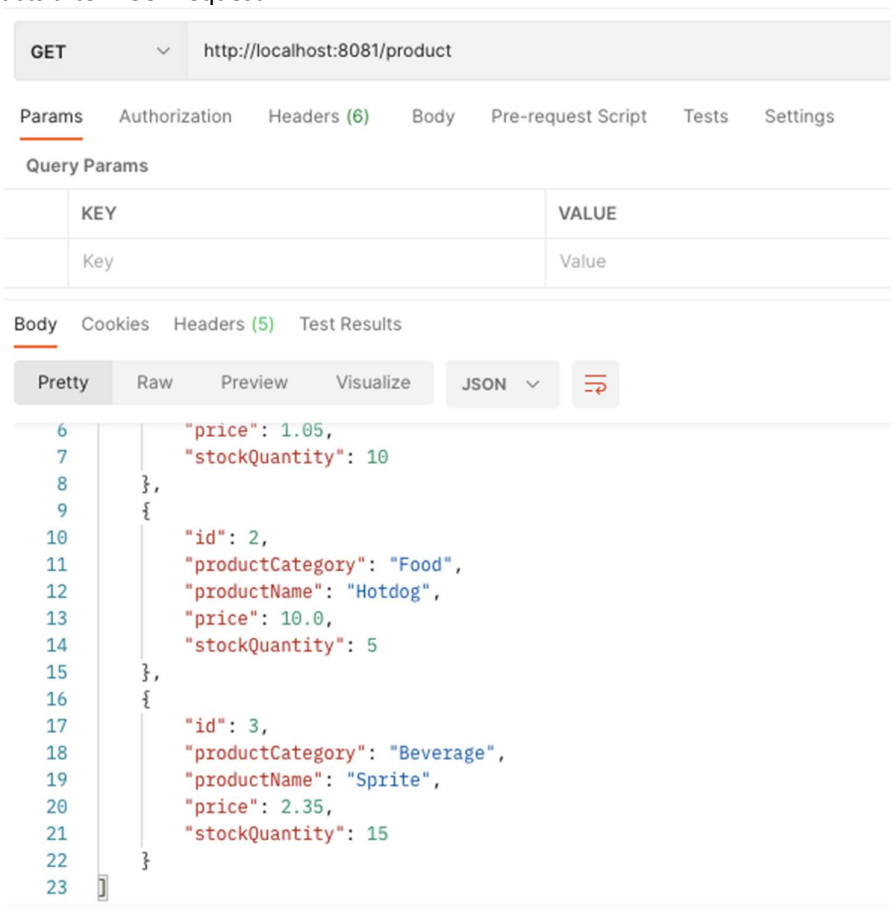


Figure 7.2 – GET Request after successful 'Create Product'

In Figure 3.3 – it can be seen that the product has been successfully inserted into the repository.

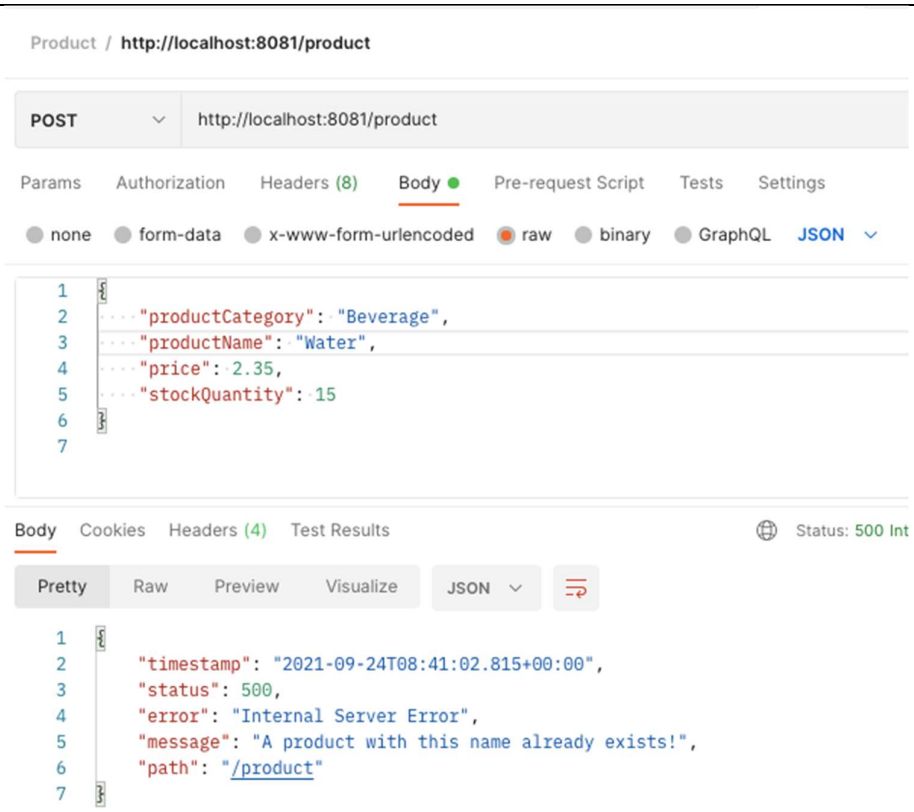


Figure 7.3 – POST Request containing same productName

In Figure 3.4 entering a new product with a productName that already exists returns an error because productName is unique.

UPDATE CUSTOMER

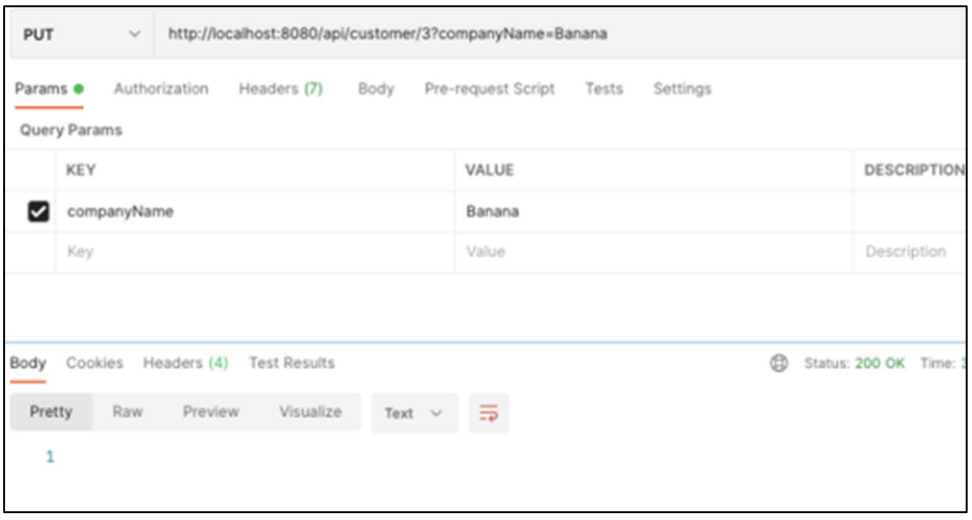


Figure 7.4 – PUT Request to customer with ID = 3

In Figure 3.5, a PUT request can be sent to the customer where id is 3. Key parameters can be used to update the companyName as seen in figure 3.5 where the key companyName is set to the value “Banana”.

```
curl --location --
request PUT 'http://localhost:8080/api/customer/3?companyName=Banana'
```

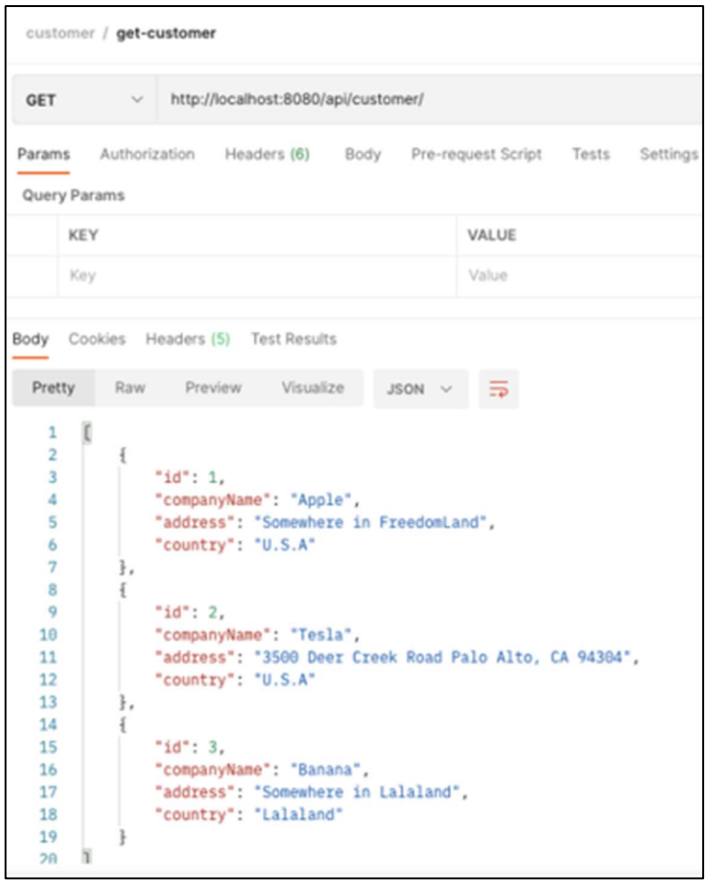


Figure 7.5 – GET Request after successful update

PUT request failed because companyName “Tesla” already exists

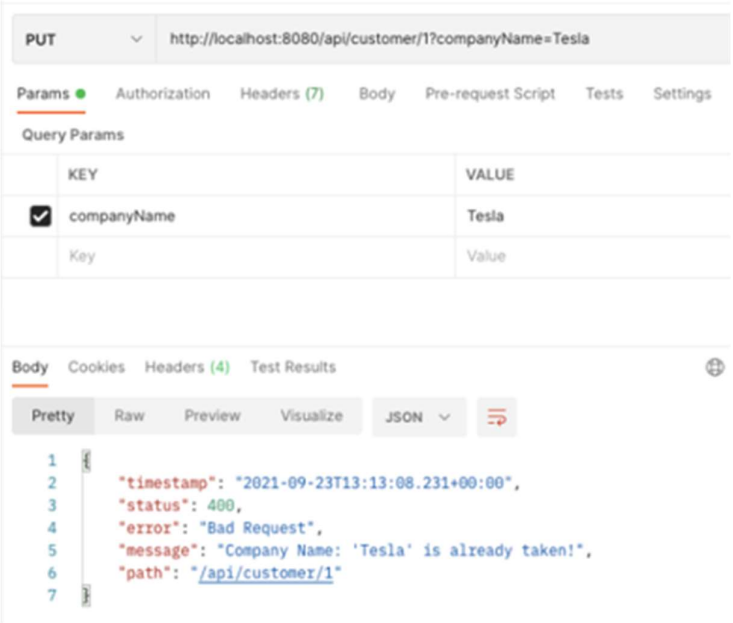
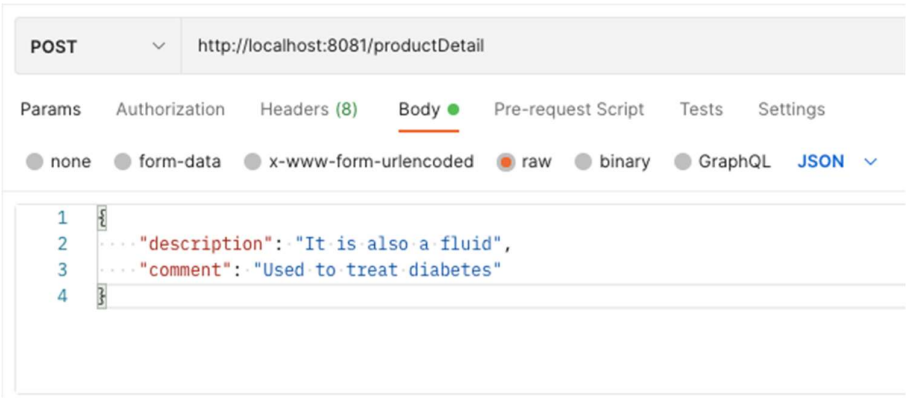


Figure 7.6– Updating customer with a company name “Tesla” which already exists in the database.

USE CASE 5: Create/update product detail

Create Product Detail

Input:
 
 	
Figure 8.1 – Creating new product detail
 	
When sending a POST request to the productDetail API of creating a product detail, a successful creation will result in no body. Neither the description or comment attributes are unique, therefore you can send duplicate values to the productDetail API.
 	
Curl:
`curl --location --request POST 'http://localhost:8081/productDetail' \`
`--header 'Content-Type: application/json' \`
`--data-raw '{`
`"description": "It is also a fluid",`
`"comment": "Used to treat diabetes"`
`}'`
 	
Output:

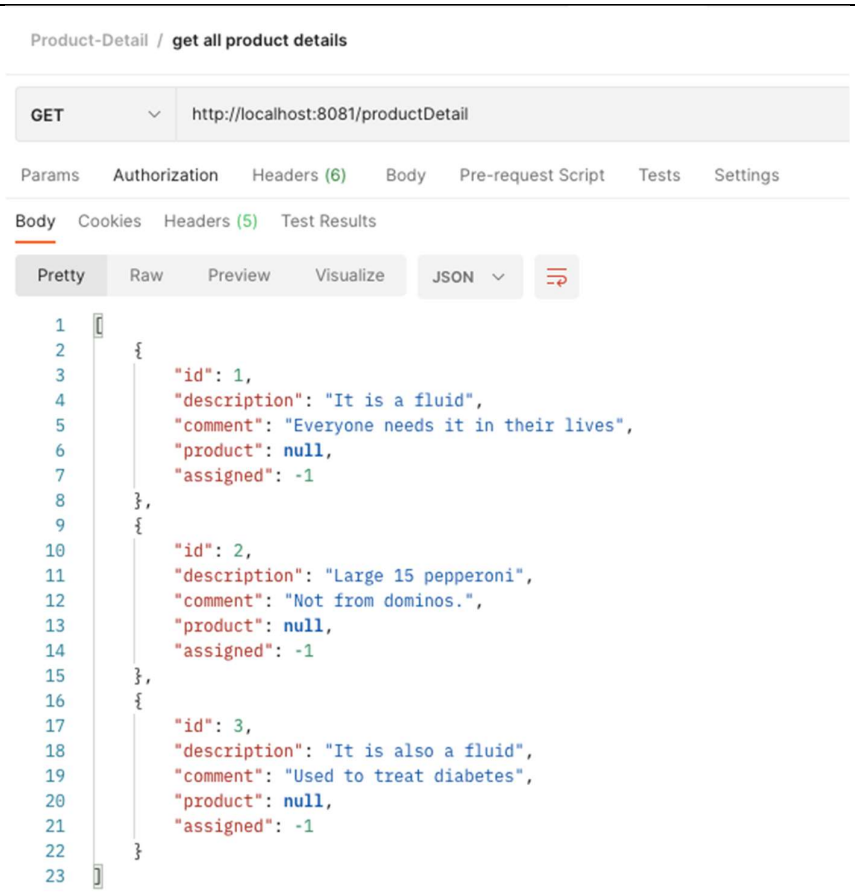


Figure 8.2 – Output of creating a new product detail

After successful POST, we can view the new entry with a GET request to the productDetail endpoint as seen in figure 8.2.

Curl:

```
curl --location --request GET 'http://localhost:8081/productDetail'
```

Update Product Detail

Input:

PUT request can be sent to productDetail/{productDetailId} using key parameters where the key value are attributes of Product Detail and values containing the value to update/assign the productDetail object.

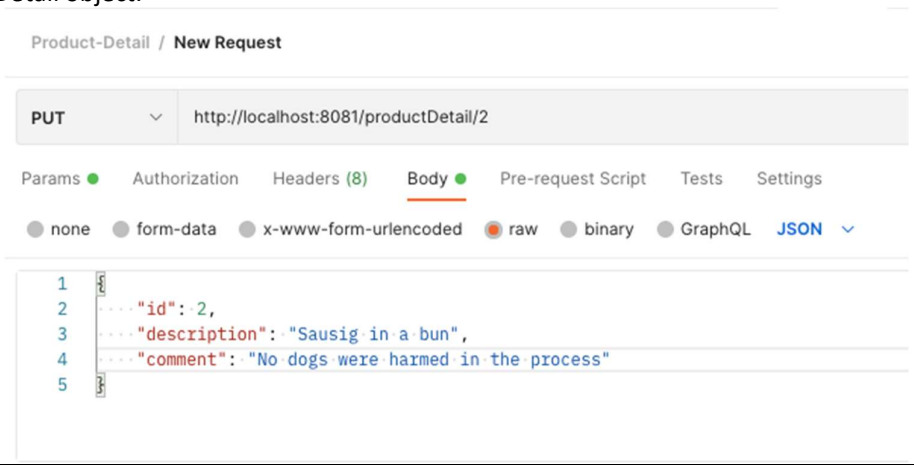


Figure 8.3 – Update request of updating a productDetail with id = 2.

Output:

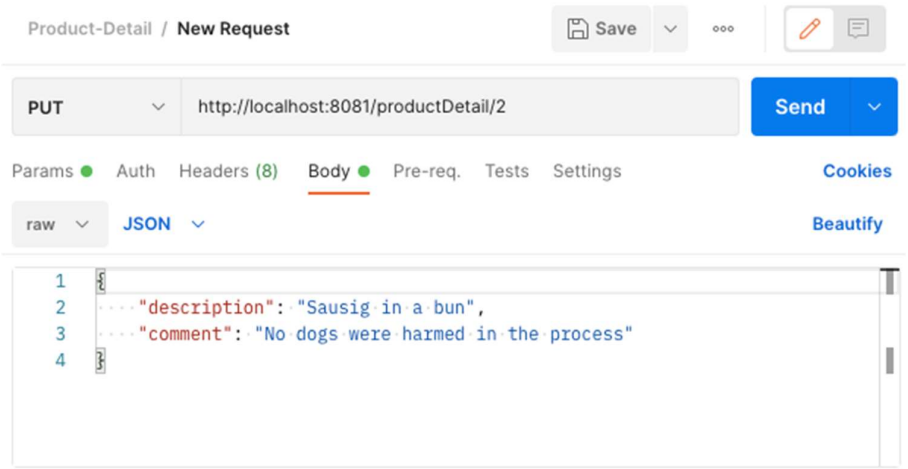


Figure 8.4 – Output of updating productDetails 2

Curl:

```
curl --location --request PUT 'http://localhost:8081/productDetail/2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "description": "This is a test",
    "comment": "This should update product 2'\''s details."
}'
```

Update product with product details

Products can be updated with adding product details by sending a PUT request to /product/{productId}/productDetails/{productDetailsId}

Where {productId} and {productDetailsId} are both replaced by a number corresponding to the ID of the objects.

Input:

```
curl --location --request PUT 'http://localhost:8081/product/2/productDetail/2'
```

This is where product with the ID 2 is assigned with the productDetail of ID 2.

Output:

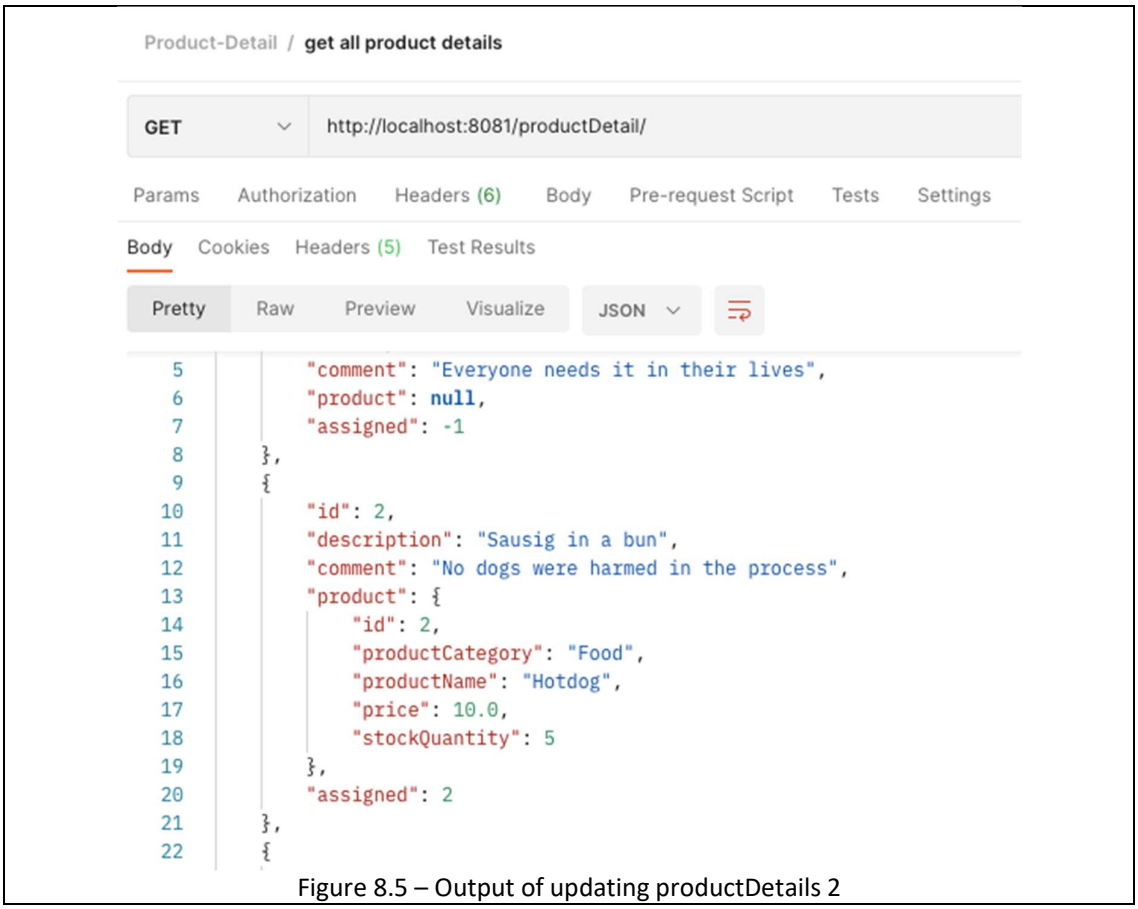


Figure 8.5 – Output of updating productDetails 2

USE CASE 6: Look up product basic info and detail

Look up product info can be done through the productDetail API. To grab all product and productDetail, a GET request can be sent to '/productDetail'.

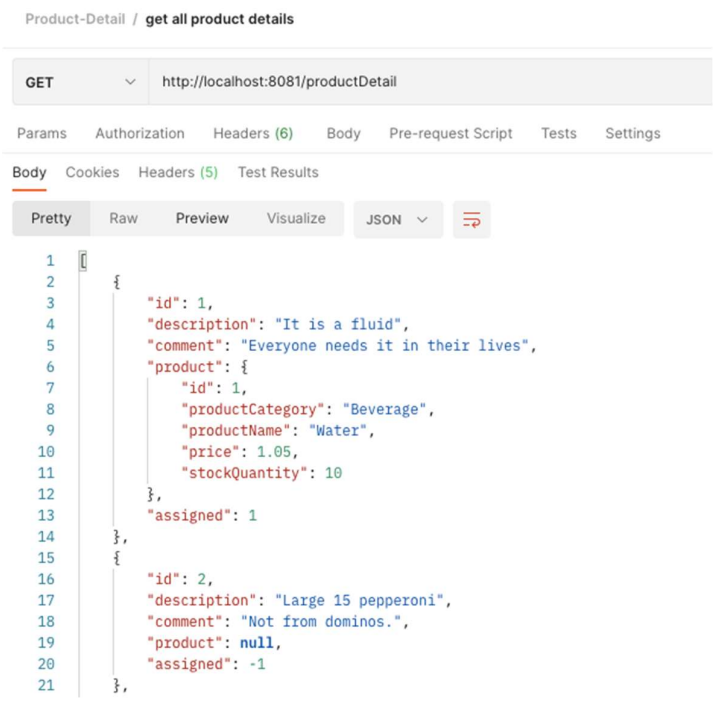


Figure 9.1 – Looking up all products basic info and product detail

or

```
curl --location --request GET 'http://localhost:8081/productDetail'
```

Looking up a specific product basic info and product detail

This can be achieved by sending a GET request to the productDetail endpoint with an ID corresponding to the productDetail ID. For example,

Curl:

```
curl --location --request GET 'http://localhost:8081/productDetail/3'
```

Output:

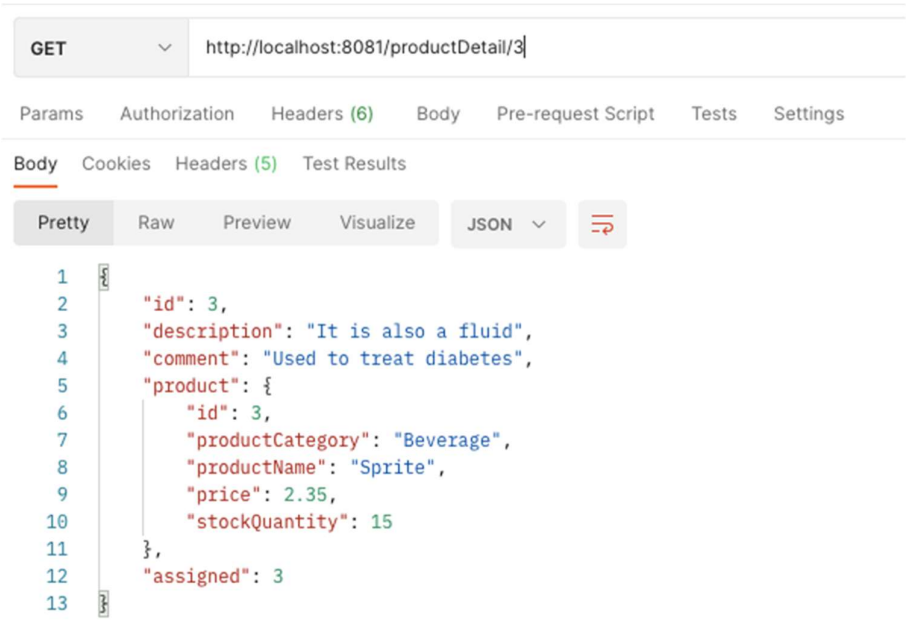


Figure 9.2 – Looking a specific product detail

Looking up a productDetailID allows us to view the product details and the product it has been assigned. If there were to be no product assigned to the product details, this will print:

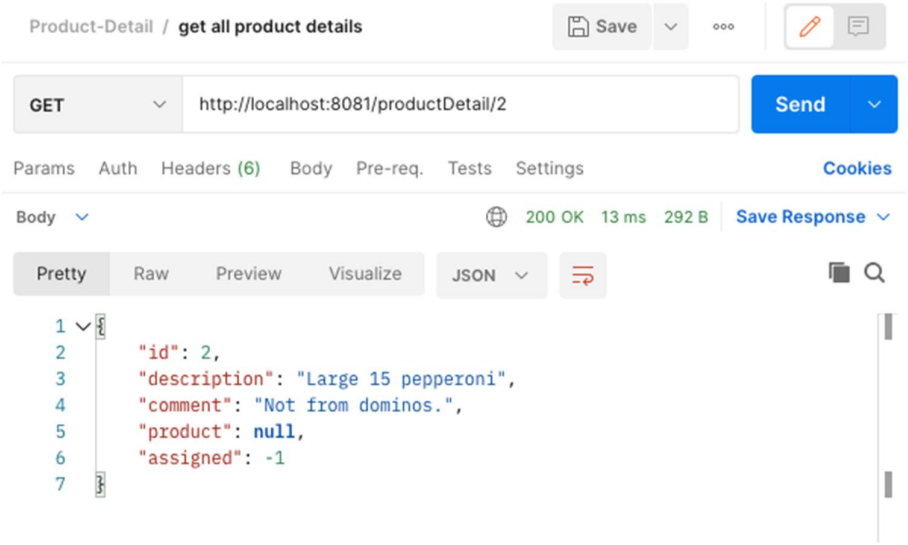


Figure 9.3 – Looking a specific productDetail with no assignment to a product

USE CASE 7: Create Order

Creating an order can be completed through the Orders API.
To create an order a POST request can be sent to ./orders with the appropriate values in the request body

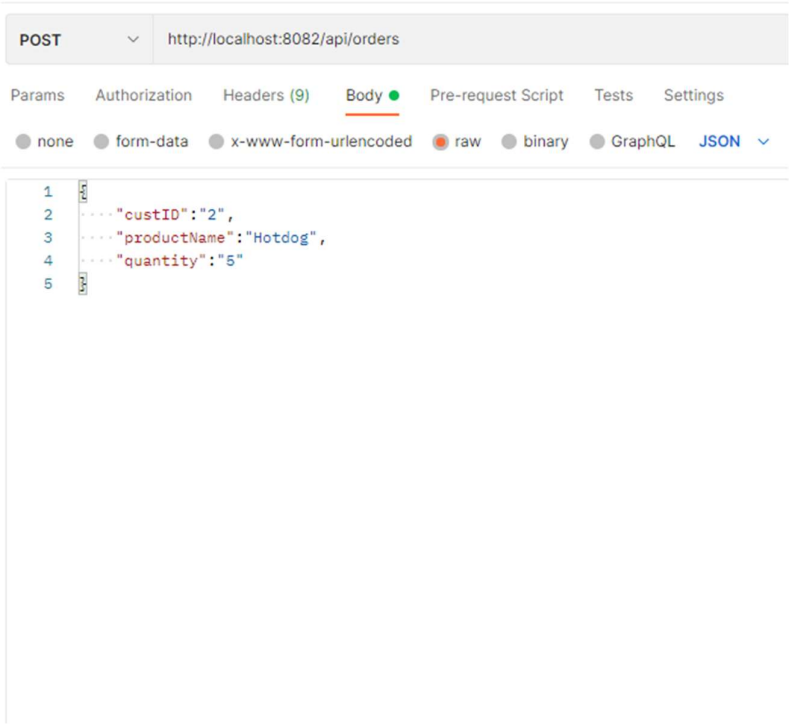


Figure 10.1– Creating a new order

Or

```
curl --location --request POST 'http://localhost:8082/api/orders' \
--header 'Content-Type: application/json' \
--data-raw '{
    "custID":"2",
    "productName":"Hotdog",
    "quantity":"5"
}'
```

This would return a '1' or true response if this was successful.
It would also appear when submitting a request for all orders as shown here

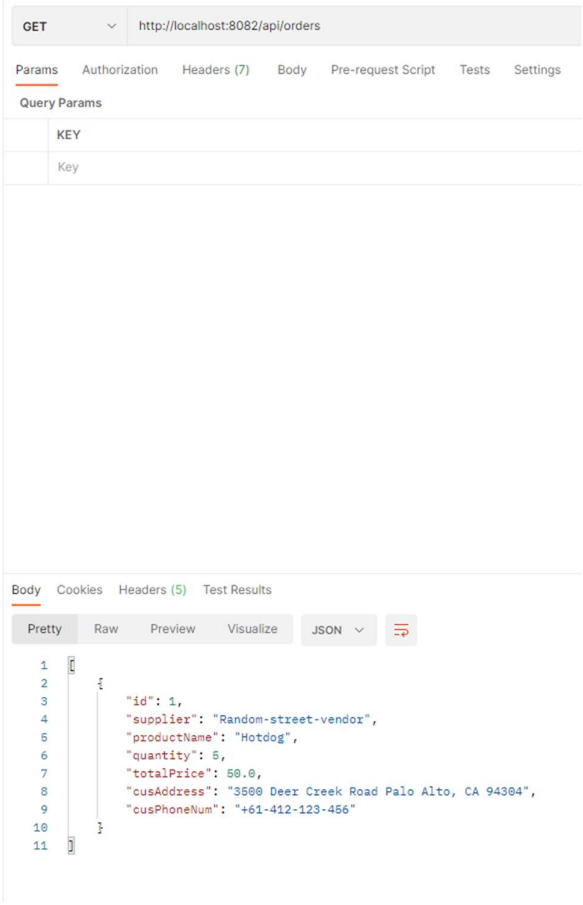


Figure 10.2 – Successful order creation

Or

```
curl --location --request GET 'http://localhost:8082/api/orders'
```

Submitting a create order request first validates a customer by submitting a request to the customer application. Upon success, it will return a customer's address and phone number

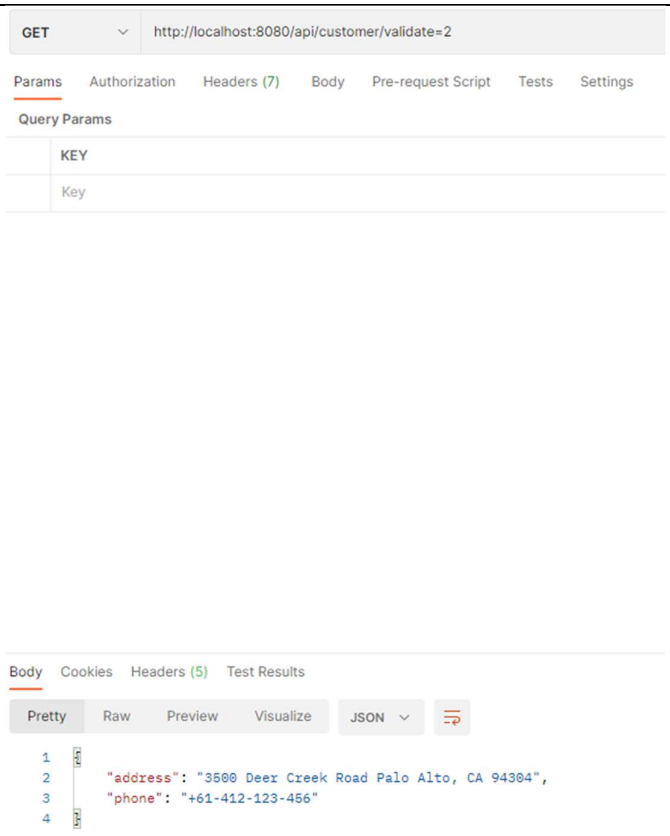


Figure10.3 – Customer Validation

Or

```
curl --location --request GET 'http://localhost:8080/api/customer/validate=2'
```

It then checks that there is enough product in stock with a request to the product application as shown here. Upon success, a unit price and supplier is returned.

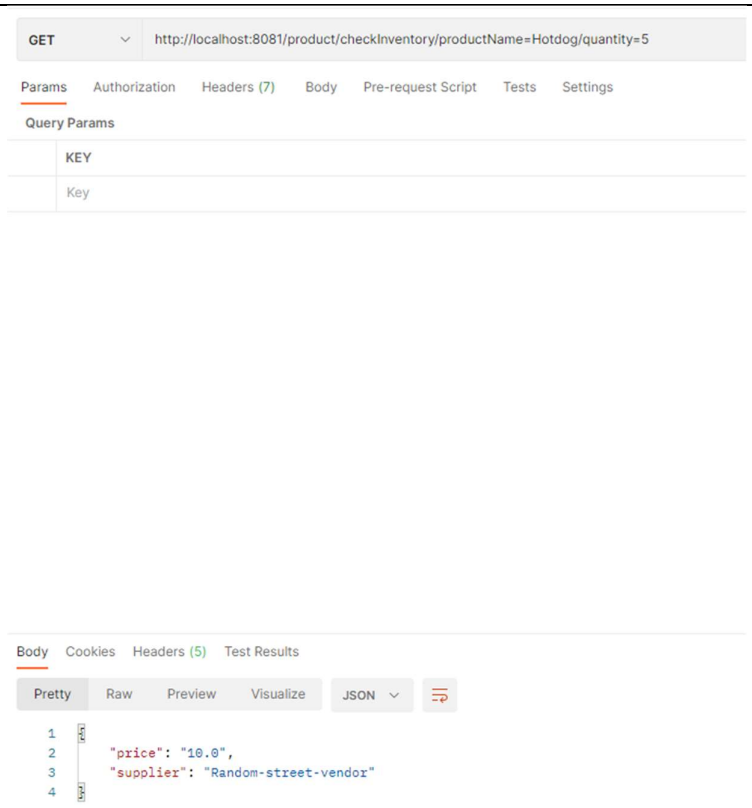


Figure 10.4 – Check Inventory

Or

```
curl --location --request GET
'http://localhost:8081/product/checkInventory/productName=Hotdog/quantity=5'
```

If this check fails, an exception will be shown


```
{
    "timestamp": "2021-09-24T10:18:46.616+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "message": "There is not enough stock for product: Hotdog. The amount of
        available stock: 0. The quantity you have requested for: 5",
    "path": "/product/checkInventory/productName=Hotdog/quantity=5"
}
```


Figure 10.5 – Failed Check Inventory

After creating an order, an OrdersEvent is then published and event listener makes a request again to the Product Application to update the inventory of the specified item. This request looks like this:

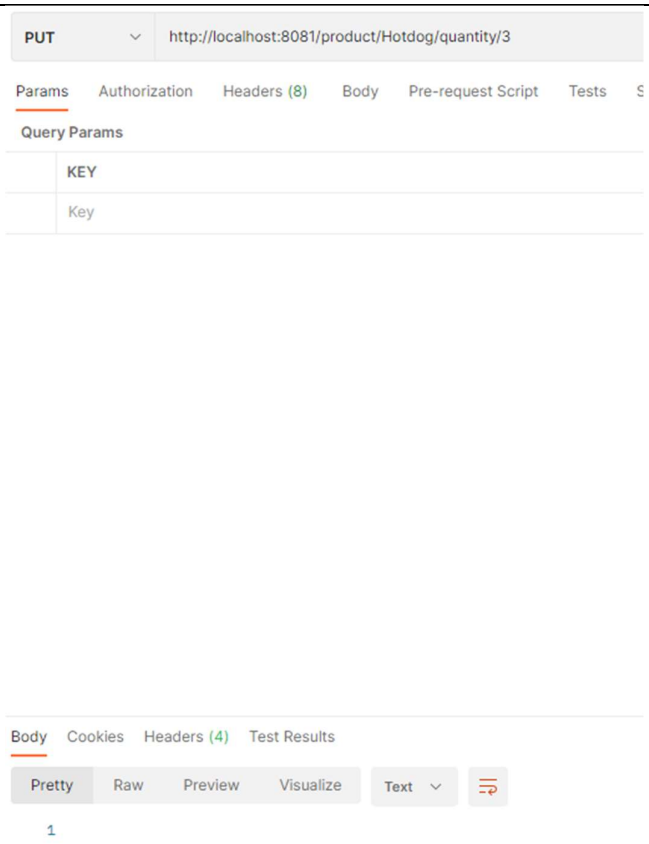


Figure 10.6 – Update Stock

Or

```
curl --location --request PUT 'http://localhost:8081/product/Hotdog/quantity/3'
```

USE CASE 8: Look up Customer basic info by order

Assumptions have been made that a customer's basic info includes only a customer's address and a customer's phone number.
Looking up this information up by order can be completed with the orders API

A GET request is made with the orderID as shown. The order being used is the same order shown in Figure 11.1

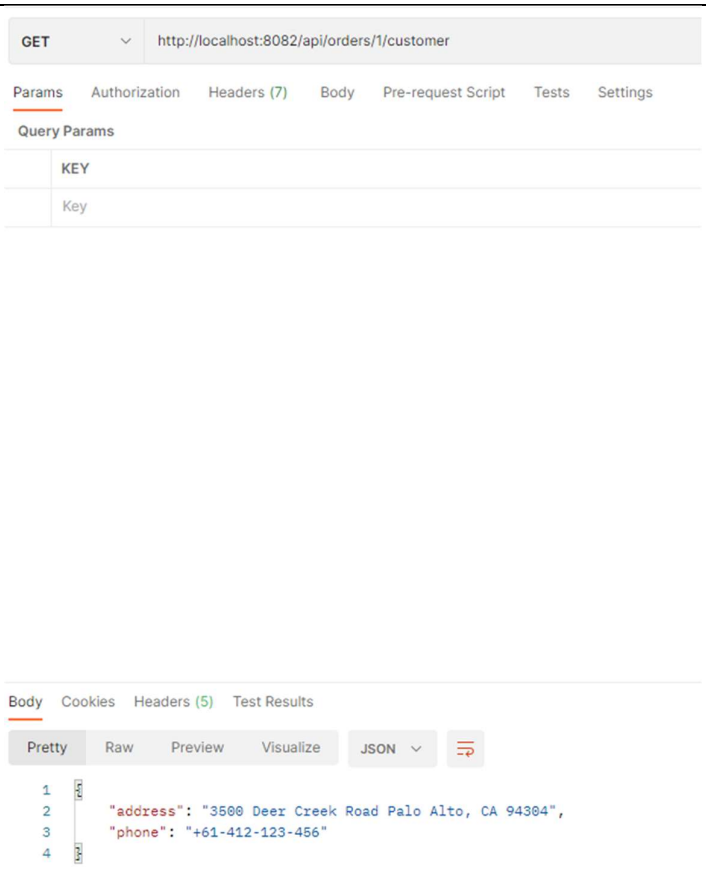


Figure 11.1 – Look up customer info by order

Or

```
curl --location --request GET 'http://localhost:8082/api/orders/1/customer'
```

If no such orderID exists, the request will receive the following response:

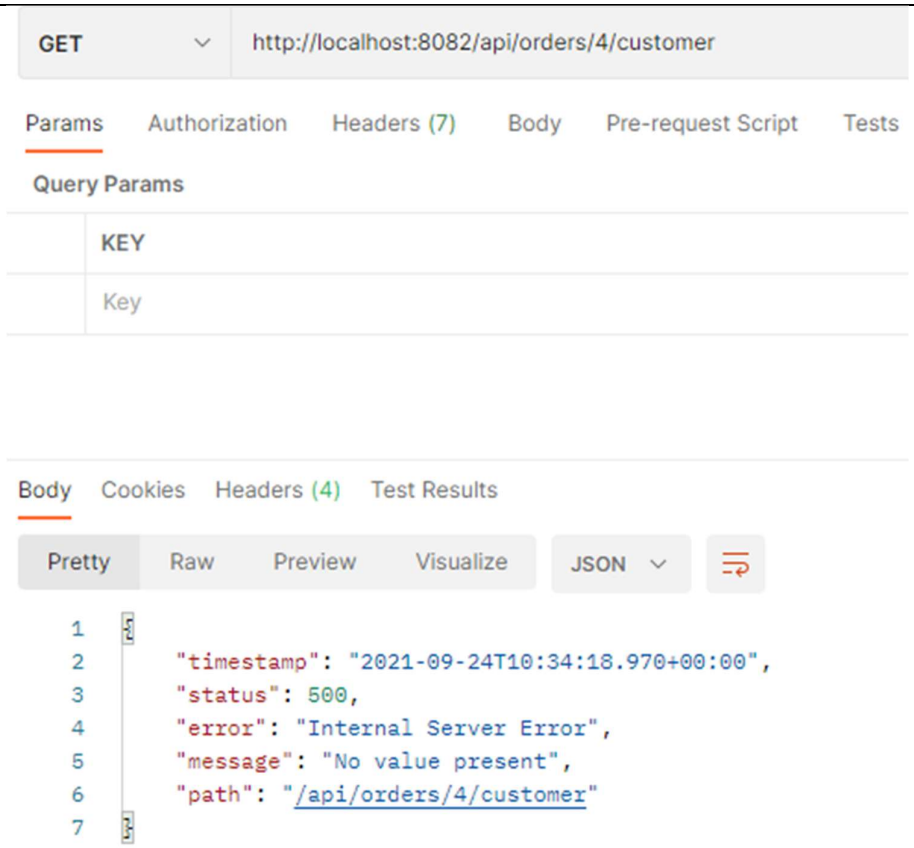


Figure 11.2– Failed Look up customer info by order

USE CASE 9: Look up Product basic info by order

Assumptions have been made a products basic information consists of a product name, supplier and unit price.
Looking up this information by order can be done by using the orders API
A GET request is submitted with the orderID as shown below. The order being used is the same order shown in Figure 7.2.

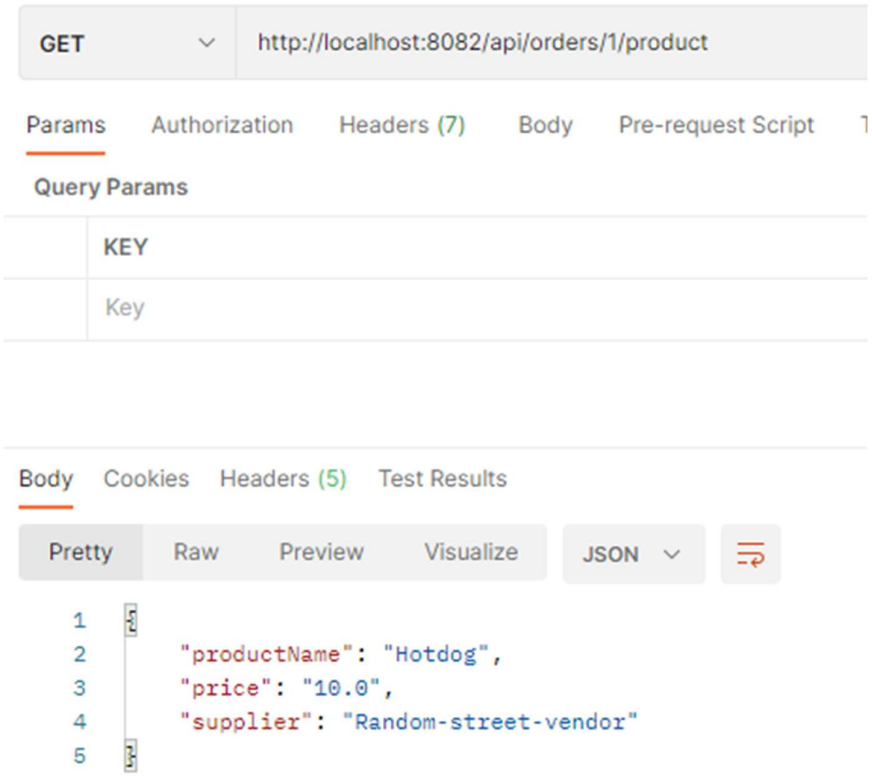


Figure 12.1– look up product info by order

Or

```
curl --location --request GET 'http://localhost:8082/api/orders/1/product'
```

If no such orderID exists, the request will receive the following response:

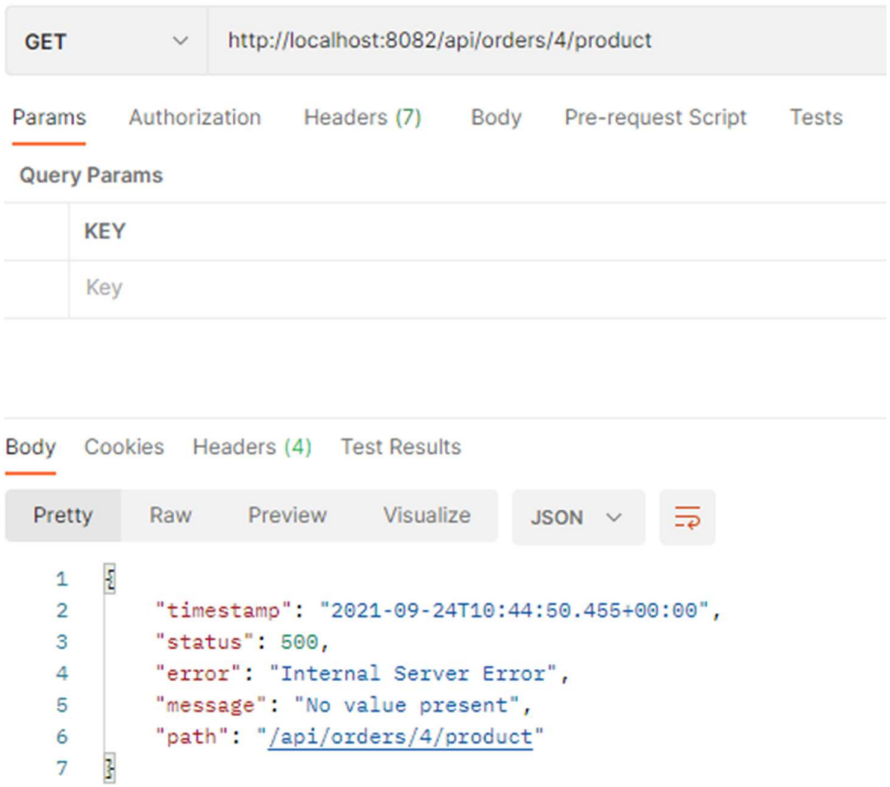


Figure 12.2– failed look up product info by order

Instructions on Building JAR files

**Note – The Compiled JAR files for all projects will also be included with the source code in the Target Directories for each project*

With the use of IntelliJ's in-built Maven package and functionality, the **target** class was built using the 'install' functionality (We did not try to attempt to build the target class via command line). The target class will also be included in the ZIP file.

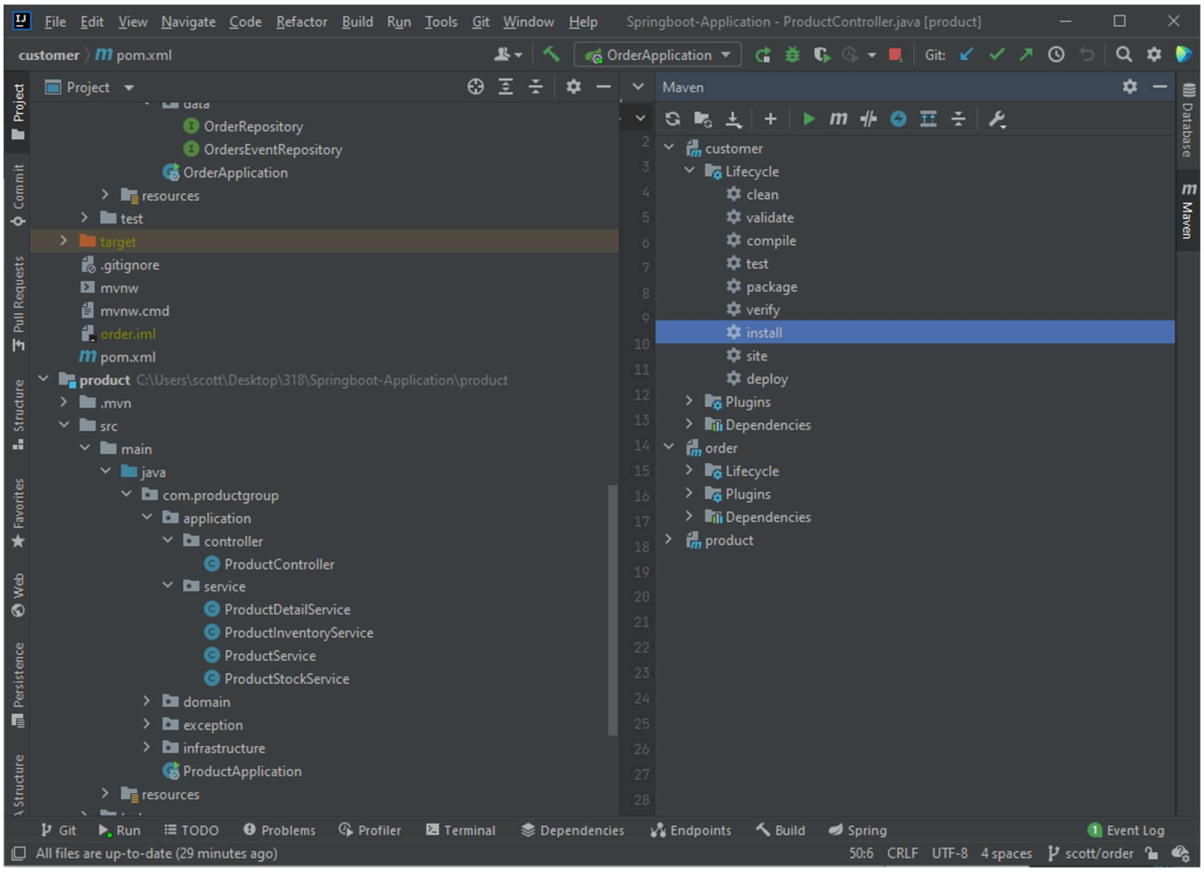


Figure 13.1 - Intellij's Maven interface

This functionality constructed the target class along with the FAT jar.

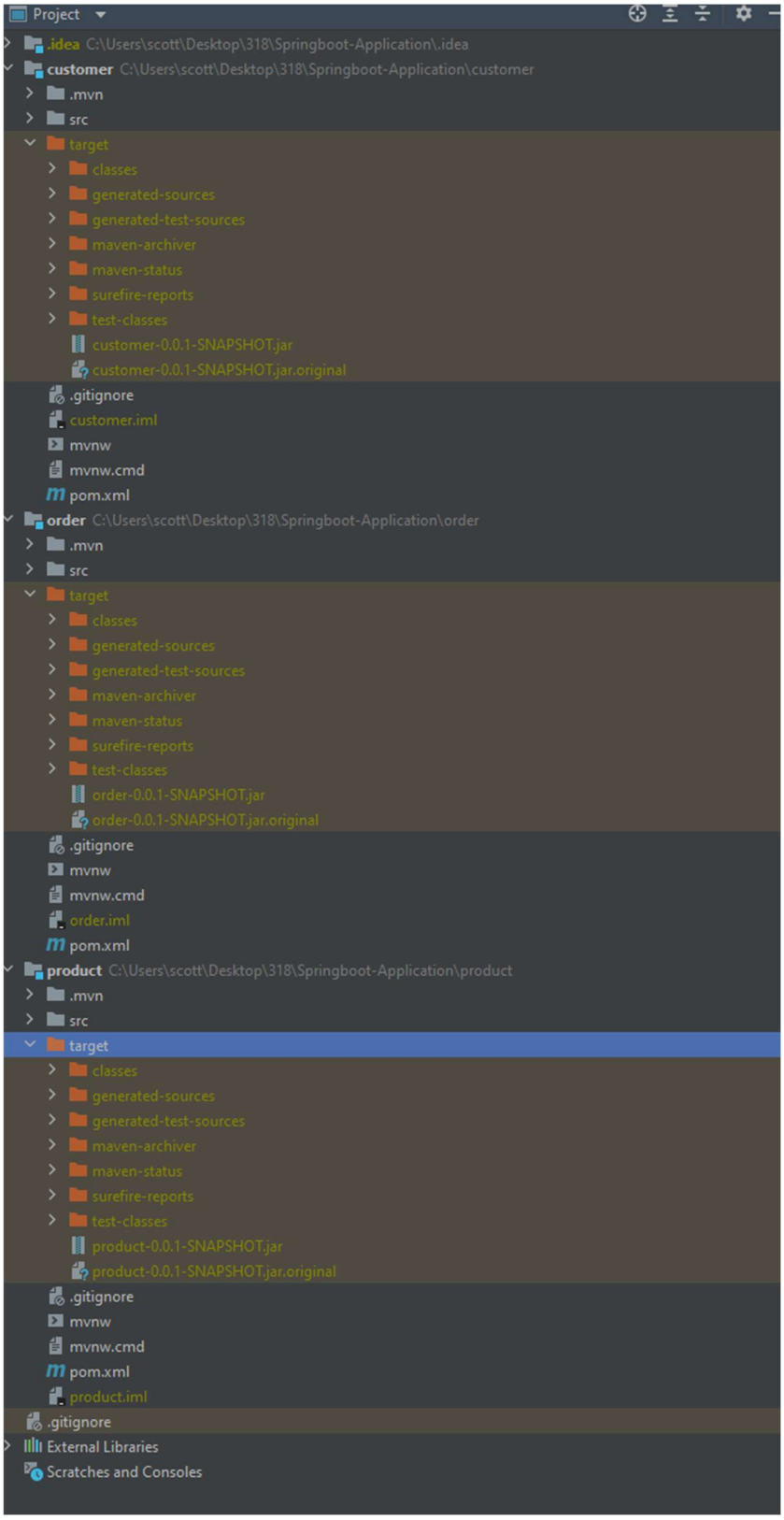


Figure 13.2 - JAR files built in Target Directories

Using Maven's install command will compile the Jar file and install it in the local repository.

Running the Jar Files

To run the 3 Jar Files, one for each project, the command below was inserted via command line whilst in the target directory for each project respectively.

```
java -jar customer-0.0.1-SNAPSHOT.jar
```

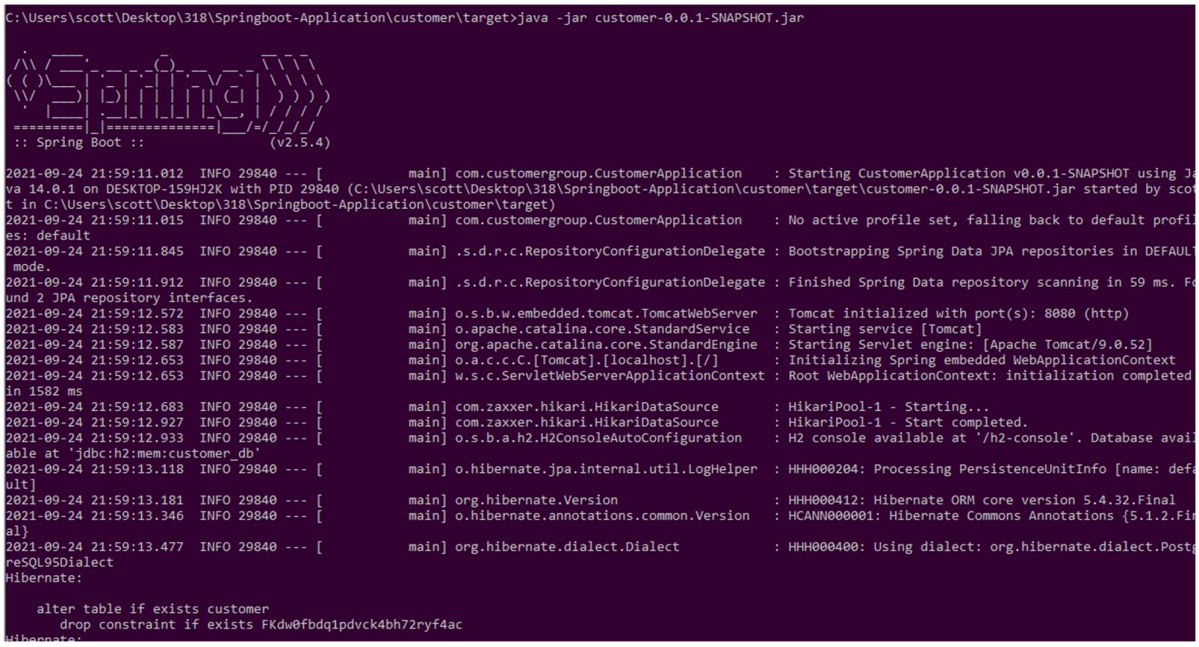


Figure 13.3 - customer application running from compiled jar file

```
java -jar product-0.0.1-SNAPSHOT.jar
```

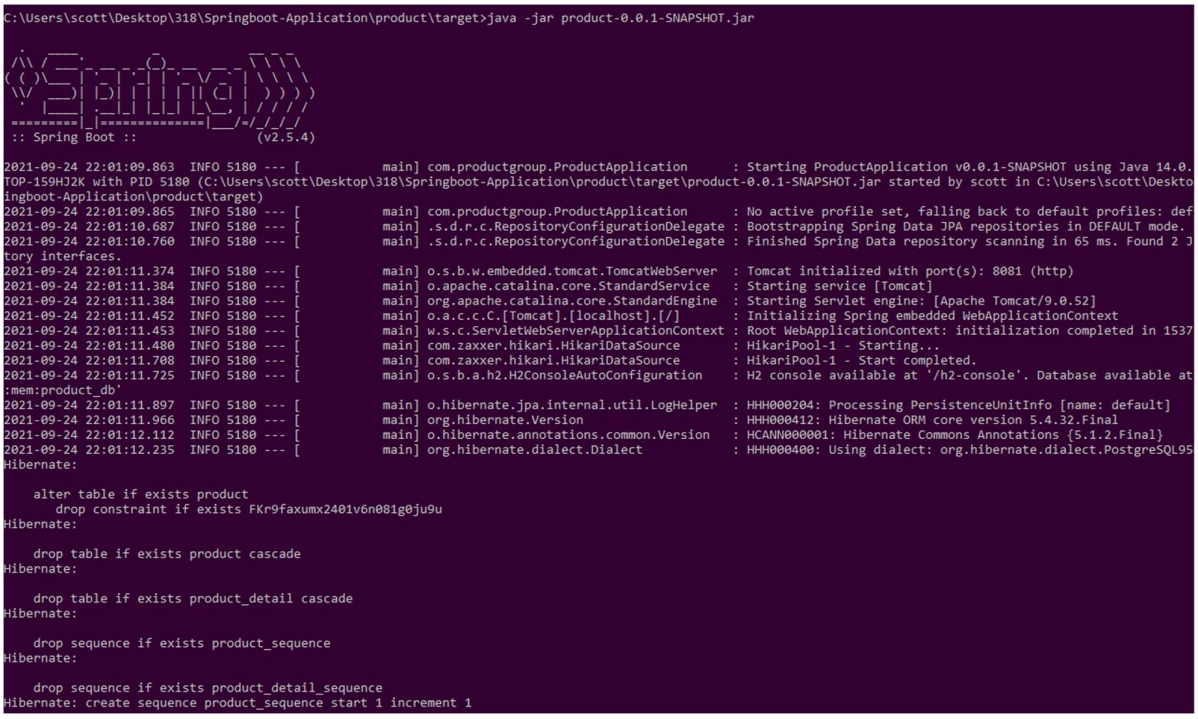


Figure 13.4 - product application running from compiled jar file

```
java -jar order-0.0.1-SNAPSHOT.jar
```

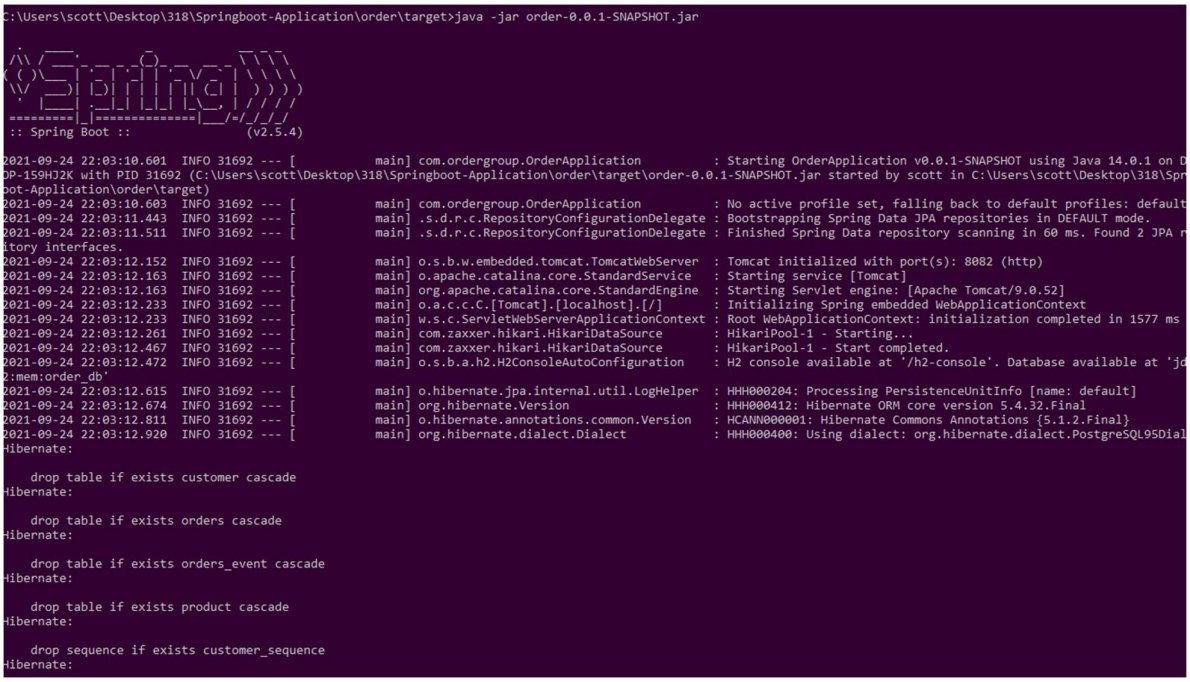


Figure 13.5 - order application running from compiled jar file

Input Requests

Input request can either be made by making requests through postman.com or by using the curl requests documented in the previous use case documentation. Further details about the usage of endpoints are also documented in the use case tables.

When running the applications, to run input requests: Replace any variables e.g {ORDERID} with the appropriate values

When running POST or PUT requests, there may be endpoints that will accept raw JSON format text. For example,

```
curl --location --request POST 'http://localhost:8081/product' \

--header 'Content-Type: application/json' \

--data-raw '{

    "productCategory": "Beverage",

    "productName": "Sprite",

    "price": 2.35,

    "stockQuantity": 15

}

'
```

Endpoints

Application	Endpoints
Customer	GET http://localhost:8080/api/customer http://localhost:8080/api/customer{CUSTOMERID} http://localhost:8080/api/contact/ http://localhost:8080/api/contact/{CONTACTID} http://localhost:8080/api/customer/validate={CUSTOMERID}
	PUT Updating a customer http://localhost:8080/api/customer/{CUSTOMERID}?{key}={value}&{key}={value} http://localhost:8080/api/customer/putRaw/{CUSTOMERID} (with inclusion of raw JSON body) Updating a contact http://localhost:8080/api/contact/{CONTACTID}?{key}={value}&{key}={value}&{key}={value}&{key}={value} http://localhost:8080/api/contact/putRaw/{CONTACTID} (With inclusion of raw JSON body) http://localhost:8080/api/customer/{CUSTOMERID}/contact/{CONTACTID}
	POST http://localhost:8080/api/customer (With inclusion of raw JSON body) http://localhost:8080/api/contact (With inclusion of raw JSON body)
Product	GET http://localhost:8081/product http://localhost:8081/product/{PRODUCTID} http://localhost:8081/product/?productName={PRODUCTNAME} http://localhost:8081/productDetail http://localhost:8081/product/checkInventory/productName={PRODUCTNAME}/quantity={QUANTITY}
	PUT http://localhost:8081/productDetail/{PRODUCTDETAILID}

	http://localhost:8081/product/{PRODUCTID}/productDetail/{PRODUCTDETAILID} http://localhost:8081/product/{PRODUCTNAME}/quantity/{QUANTITY}
	POST http://localhost:8081/product
Order	GET http://localhost:8082/api/orders http://localhost:8082/api/orders/{ORDERID}/customer http://localhost:8082/api/orders/{ORDERID}/product
	POST http://localhost:8082/api/orders