Requirement 1: Identification of the problem you are trying to solve by building this particular app:

In March of 2020, I decided that I wanted to take a leap of faith and build my own gaming computer. This was an exciting decision but it was also a very intimidating decision. As I quickly found out, the amount of computer parts available online seems almost infinite. The first problem that I ran into was compatibility, sure these parts all looked really cool and they were all self-proclaimed great parts, but how was I supposed to know what I needed? Or if any of these parts would even work together? Would they all fit into the same case? Were these parts worth the price they were asking for or were they merely brand name parts with egregious prices?

I quickly found out that an insane amount of research was going to be needed. The second problem I ran into was the naming conventions companies use to name their products. Take AMD and Intel CPUs for example. What even is a Ryzen 7 3700x anyway? Or an Intel i9 12th gen 12900k?

Let's say you decide on a CPU, great! Next step, maybe you want to look at graphics cards.

Now you have another confusing decision to make, Radeon RX 6900 XT? GeForce RTX 3080 Ti?

For a novice, hell, even an intermediate enthusiast, these numbers can often be meaningless. Let's say you make the smart decision and bring up the specifications of both of those graphics' cards.

Instantly you're greeted with specifications you've never even heard of, memory bus? Core clocks? Cores? Power consumption? Do I want higher memory bus or lower memory bus, what about the core clocks I just saw? Surely a bigger number is better? Power consumption? Why is that even relevant, wont my computer be plugged into an outlet?

These are all decisions and problems I faced and needed to overcome. Finding some modicum of confidence was potentially the trickiest part. In the end, I built my computer and it worked spectacularly for some time, eventually my graphics card developed an issue with drawing current and consequently it died and almost bricked other parts at the same time. By this time, I had owned the PC for 2 years, replacing a graphics card felt like child's play now. Eventually, I decided a 3070 was going to be a terrific upgrade from my previous 2080 super, though I was very aware that the 30 series cards were significantly larger than the 20 series. I used a website called PC Part Picker, the use case of the site is to check compatibility, size clearances of hardware. The website gave me the green light and I made the purchase. Low and behold, the 3070 didn't fit. It is also worth mentioning that my PSU covered the increased power draw by a number that is not nearly as large as it should be.

Problems Solved by my API and its current use-case:

Currently, my API contains a series of accurate information. This information is easily accessible through GET requests. More information can be posted to the API using the POST method and information can also be altered using the PUT method. Information can be deleted, however some of the products are hard coded in to maintain and protect relationships between tables.

Due to the importance of information accuracy, PUT and DELETE methods are protected and require administrator access. Some models also have POST method protection as well.

As it stand now for presentation, my favourite features of the API are:

- Find a list of products in convenient location, even if that data originates from another table
- Compare the voltage requirements of a GPU to the voltage provisions of a PSU.(e.g.,750w needed, 850 provided)
- Compare the socket type of motherboards to the CPU type (e.g., AMD CPU = AM4 socket)
- Find Ram, its storage capacity and its memory type
- Find mock up ratings of the API experience
- Find ratings of each individual part in their own respective tables

Requirement 2: Why is it a problem that needs solving?

As a result of everything explained in the previous section, I quickly found out I needed my own catalogue and compatibility checker for hardware. With this API, I am able to check if parts are compatible, recommend parts to friends and family and also know early on if their existing hardware will have the capacity to support these parts.

This in turn will reduce the complexity in parts research and also provide a reassurance on the compatibility of parts. The API also offers an unbiased rating system which in turn provides a more reliable rating.

Computers are such a vital part of everyday life in the modern world. Right-to-repair laws allow people to repair and replace parts on their computers and laptops. Enthusiasts also enjoy replacing, upgrading and collecting parts. With these 3 points considered, the need for an API that can provide quick, clear and reliable information seems greater than ever.

Another reason I feel passionate about is the inclusivity of computer building between generations. As of right now, computers are an absolute mystery to my parents and grandparents. This is the case for a large number of families in the world and the growing complexity of computers further drives that generational gap apart. By providing simplified, yet accurate information, this allows people from older generations that do not have a technical background to still get involved and hopefully find the right parts for their specific needs.

Summary of the problem and why it needs solving:

The computer market is flooded with an insane number of parts, those parts in turn are all subject to outrageous naming conventions and branding. Those same parts are further priced unfairly in some cases and often do not provide accessible information in regards to compatibility. This means that a large amount of work is needed to build a computer and it further removes accessibility to a staggering percentage of older (and even younger) generations.

In order for computer repairs and builds to remain widely available to the general public, there is a need for easily accessible, accurate and reliable information.

Requirement 3: Why have you chosen this database system. What are the drawbacks compared to others?

I chose PostgreSQL as my database system for a few reasons.

- OS compatibility
- Data Type compatibility
- Data Integrity support
- Security
- Extensibility
- Scalability

OS compatibility:

PostgreSQL is compatible with all major Operating Systems such as Windows, Linux, macOS and UNIX.

This allows for incredibly valuable scalability and future proofing. Should API development continue past submission, I will be able to continue developing this API knowing that if I ever need to continue working on it on another computer, or if I ever collaborate with another developer, PostgreSQL will allow that to happen seamlessly.

Data Type Compatibility

In its current state, my API only utilises primitive data types such as Strings, Integers and Booleans. It also relies on the JSON document data type as a means to present the information present in my database. PostgreSQL itself however is compatible with an outstanding number of data types, some of which will almost certainly be utilised in future developments. In particular, PostgreSQL supports structured data types such as arrays, date/time and goes a step further in providing its own predefined dictionaries for multiple languages. This adds tremendous upgrade options to my API.

Data Integrity Support

Data integrity is offered through multiple reliable channels. The main ones I have utilised are the Primary and Foreign Keys, Unique Constraints and Not-Null constraints. Some Check Constraints were implemented through my models (such as correct data types, Integer returns True.)

I have 12 models in my API, those 12 models are further defined by 12 corresponding schemas and 12 corresponding controllers.

Each of the 12 models has its own unique Primary Key, you will find that key described as ID in the models. (Customer_ID = Primary Key for example.)

The relationships are then built through Foreign Keys, schema definitions (such as nested fields, established relationships (backref etc))

Unique constraints are found in the form of Primary keys, these two constraints collaborate to provide a unique identifier. The ID of an item auto increments in each relation. That ID is related to that item and that item only, even after deletion, that ID will not be reused.

Not-Null constraints were originally defined as “nullable = False”, however they were later indirectly implemented into schemas for POST and PUT methods. You will find them in the schemas defined as either “required = True” or in the form of validation. (i.e., validate(length min=1 max =10)

Nullable = False constraints can still be found in the Admin model.D

Security

PostgreSQL has security via a number of avenues but the one that I liked the most was the SSPI. PostgreSQL is protected by SSPI (Microsoft Security Support Provider Interface). This one was the most relevant security measure to me as my computer is also a Windows Computer. SSPI is the foundation for Windows Authentication, SSPI provides a means to carry authentication tokens between the client and server computer. PostgreSQL uses SSPI for secure authentication with single sign-ons. Single sign-ons are used to reduce attack avenues from malicious parties, it achieves this by reducing log ins to one set of credentials. User’s login only once each day and only use one set of credentials, this reduces the amount of credentials available for attack and increases enterprise security.

Extensibility

Extensibility is found in these channels:

- Ability to create/define and use custom/new data types
- Design custom functions
- Use code from multiple programming languages without recompiling database

To be honest, I used none of these. But it's still an impressive feature and having the ability to test these things out is never a bad thing.

Scalability

PostgreSQL scales really well, it can handle enormous chunks of data and will benefit on all the extra resources your computer provides to it (such as better CPU or RAM). While I haven't deployed my database anywhere, if I was to scale it up to a significant amount of data, I would have a number of deployment options such as Cloud SQL (Google Cloud Platform) or even AWS (another popular Cloud Service).

Scalability is quite possibly one of the major decision factors in the real world, for this assessment, I knew I wasn't going to be creating and storing too much data, real world implantations however could see PostgreSQL managing terabytes of data.

Due to this flexibility, it became a solid first choice as my data base.

Requirement 4: Identify and discuss the key functionalities and benefits of an ORM

An Object Relational Mapping approach allows a user to take queries and manipulate data from the database. The intent of this technique is to convert data using object-oriented programming approaches so that our data can be represented as a virtual object in our database. In my database, my models defined virtual objects in my database. By creating instances of these classes, I get objects. As a result, I can easily perform a number of operations on those objects to return or manipulate information.

```
#! username and email must be unique when creating an admin.
class Administrator(db.Model):
        __tablename__ = 'admin'
        admin_id = db.Column(db.Integer, primary_key=True)
        username = db.Column(db.String(), nullable=False, unique=True)
        email = db.Column(db.String(), nullable=False, unique=True)
        password = db.Column(db.String(), nullable=False)
```

In the picture above, I have defined a class which will be used to create instances in my database, these instances will further evolve to become my objects.

In the picture below, I am creating an instance of the class Admin. Once this instance is seeded to the database, I will have my first object.

These objects act in accordance with the MVC architecture of the API.

```
#CREATE ORIGINAL ENTRY FOR ALL TABLES. REMAINDER OF THE DATA WILL BE ADDED VIA POST
def seed_db():

    admin1 = Administrator(
        username = 'Chris',
        email = 'chris@admin.com',
        #encrypt password
        password = bcrypt.generate_password_hash('Weeeeeeeee').decode('utf-8')
    )
    db.session.add(admin1)
```

The purpose of ORM is to design an object-oriented layer between my relational database and object-oriented-programming. Doing this provides me with the power of SQL without actually needing to use SQL. Mapping describes the relationships between my data and my objects without needing to understand the structure of the data. Now, I know that last sentence seems counter-intuitive, and for the original developer, they will still need to understand the data structure. However, this approach is designed to incorporate tremendous amount of SQL power into the program without needing its user to understand SQL and that is where the benefit lies.

Benefits of an ORM

Productivity

With a small amount of planning and setting up, ORMs afford developers and users powerful data manipulation. The increased productivity is found once the program has been established. Once the architecture of an ORM application is defined, SQL queries are executed automatically when called upon. It also adds accessibility to the application, allowing users to query and manipulate data without a complex understanding of how its all working.

Productivity is also found in the avenue of debugging, due the way ORMs are designed, every object in the database can be queried or manipulated. So, when data starts behaving incorrectly, it is easier to understand and locate the issue.

For example, say you have a user object, that user has some details such as a first name and a last name. Let's say you've also designed a query that checks the user table and if it finds users with first names, it returns them. Now let's pretend it's not working. With an ORM, our debugging methodology is actually rather simple.

That bug could be occurring in 1 of 3 places:

- Our query may not be designed correctly
- Our users table may not have any objects in it *or* the users may not have first names
- Lastly, we may have not designed our instances in accordance with our models

By understanding these avenues, developers can quickly track down the problem area and resolve it.

Situations like the one above are fictional, though quite similar to the ones I experienced personally.

Application design

To define a good ORM, we need to adhere to best practises in a number of other areas. For example, if we want all the parts of our application to work correctly, we need correct and best practices in regards to ORM, MVC, naming conventions, DRY coding, accurate and useful imports etc.

By acting in accordance to these practices, we are forced to create good, functional programs. Good, functional programs allow for scalability and extensibility. With good application design and planning, we can reduce testing and debugging efforts initially and later down the track. There are **no** drawbacks to following best practises.

Code

Once an application is designed, the code created for that application can be used again in future application via creating a class library. This means that as time goes on, ORM and MVC designs maintain and actually increase flexibility and efficiency.

Summary

ORM affords developers the power of SQL while also reducing the amount of work needed in the long-term. Once objects are designed, SQL queries can be designed to return a number of different manipulations of data. For example, someone without SQL knowledge will still be able to perform the command " SELECT * FROM customers;" via the GET method defined in the controller.

Requirement 5: Document all endpoints for your API

Total List of all Endpoints + 1 Query String for Products

Below are all of the possible links found on my API, in the links 'x' represents an Integer ID that may be passed to the URL alongside different HTTP Verbs.

localhost:5000/admin/register

localhost:5000/admin/login

localhost:5000/admin/x

localhost:5000/cpu/

localhost:5000/cpu/x

localhost:5000/customers/

localhost:5000/customers/x

localhost:5000/gpu/

localhost:5000/gpu/x

localhost:5000/compatibility/

localhost:5000/compatibility/x

localhost:5000/motherboards/

localhost:5000/motherboards/x

localhost:5000/orders/

localhost:5000/orders/x

localhost:5000/products/

localhost:5000/products/x

http://127.0.0.1:5000/products/?description=ASUS Prime A520M-K. (AMD)

localhost:5000/psu/

localhost:5000/psu/x

localhost:5000/ram/

localhost:5000/ram/x

localhost:5000/ratings/

localhost:5000/ratings/x

localhost:5000/voltages/

localhost:5000/voltages/x

Controllers:

My application has 12 controllers each containing up to 5 end points. They are listed below:

ADMIN

The admin controller contains 3 endpoints. None of these end points need a token to function.

Endpoint 1:

Request VERB = POST

Request ADDRESS = localhost:5000/admin/register

Functionality and Required Fields:

This endpoint is used to register a new admin into the API. In order to register a new admin, you need 3 fields:

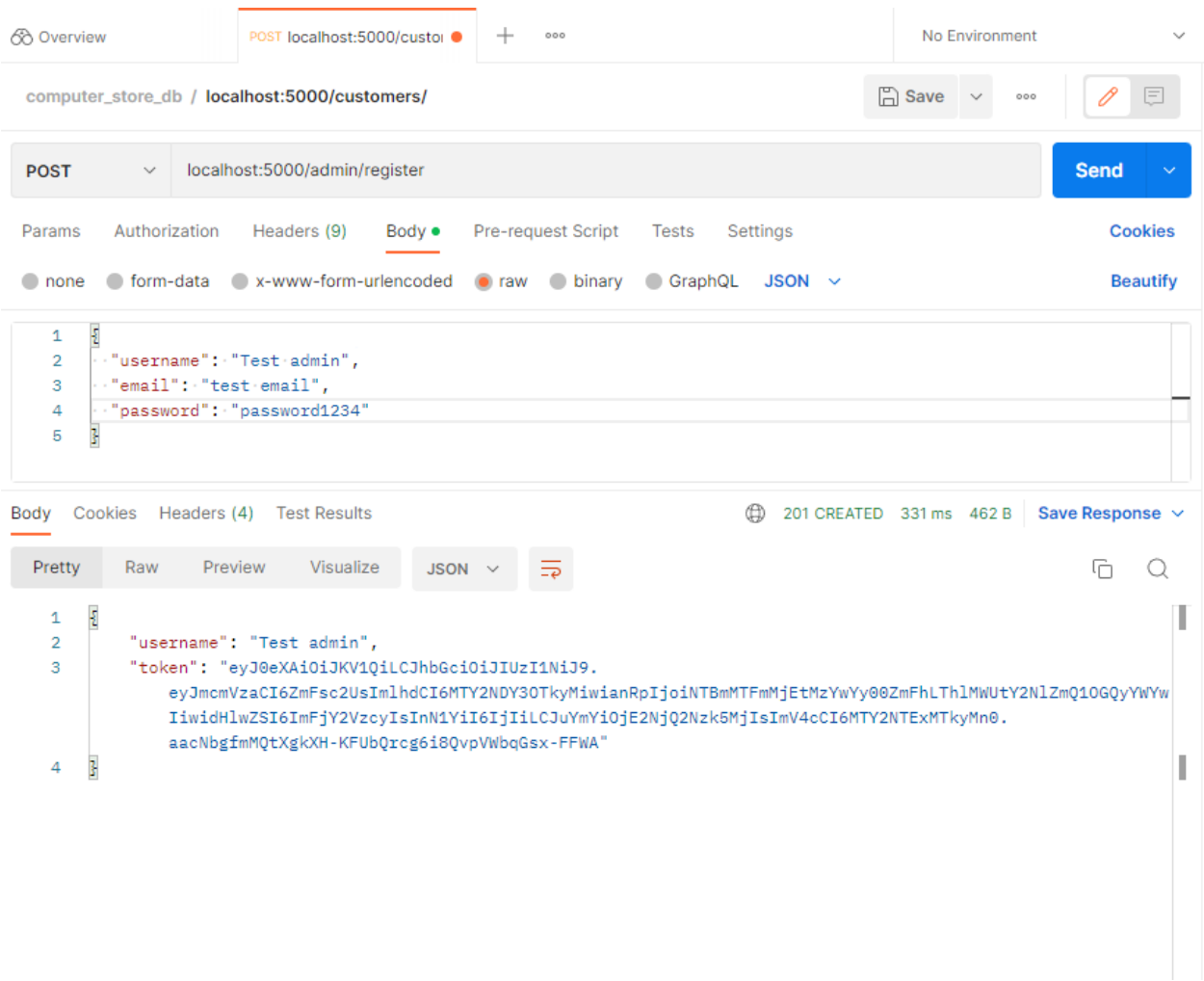
Username: Username must be a string

Email: Email must be a string

Password: Password must be a string; it must also be a minimum of 8 characters.

Expected return

If you provide this request with sufficient fields and correct data types, it will return the username and the token for the new admin that you have created



Endpoint 2:

Request VERB = POST

Request ADDRESS = localhost:5000/admin/login

Functionality and Required Fields:

This endpoint is used to log in to an existing admin account to receive the token needed to gain access to some JWT protected end points. To login, you need 3 fields:

Username: Username must be a string

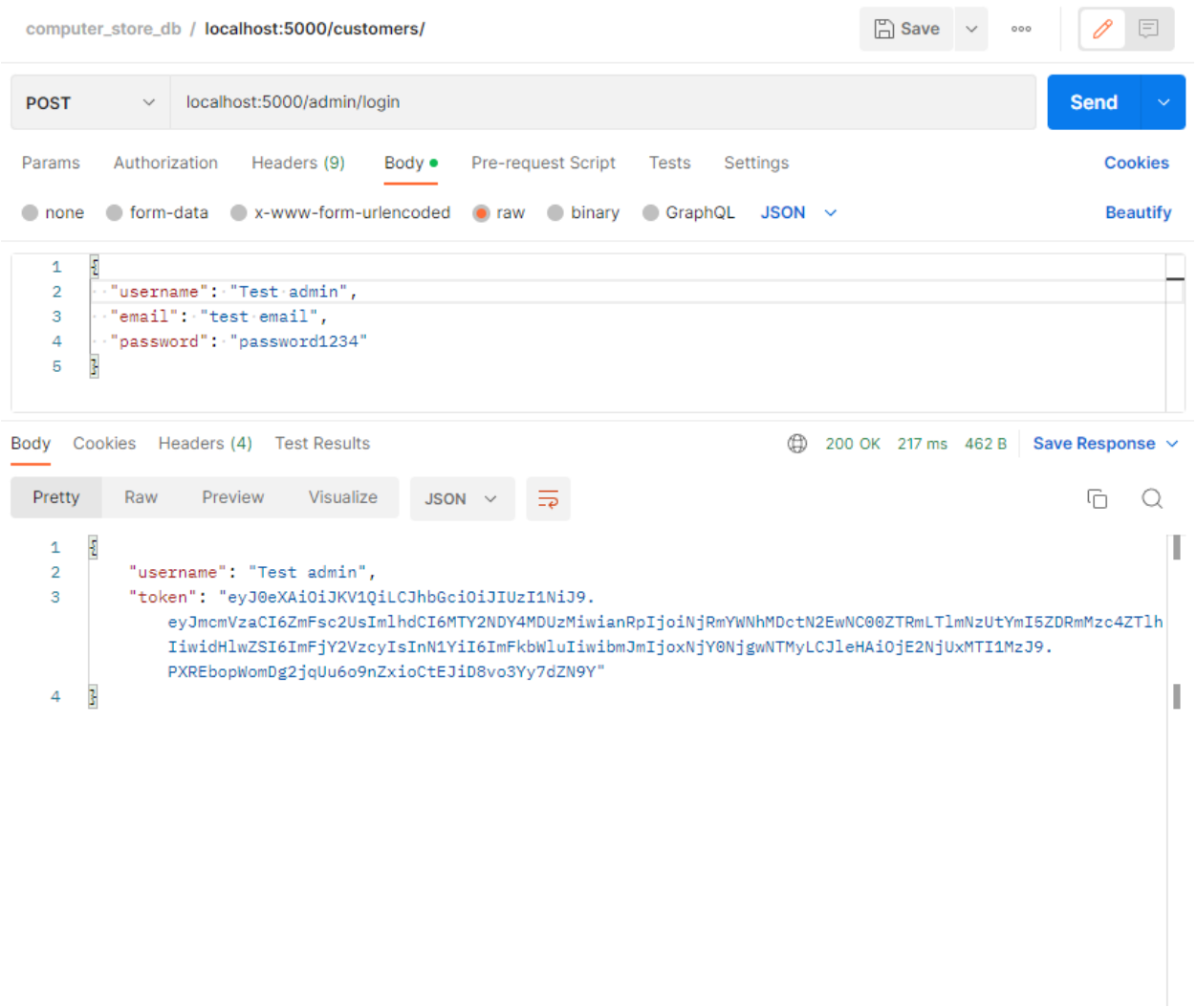
Email: Email must be a string

Password: Password must be a string; it must also be a minimum of 8 characters.

NOTE: These details need to match the details you used to register the admin.

Expected return

If you provide the same details you used to register the admin, it will return your token. To gain admin access, you need to take that token and place it into the authorisation header. Select bearer token from the drop-down menu and then provide the token and you will be able to process protected admin endpoints.



Endpoint 3:

Request VERB = DELETE

Request ADDRESS = localhost:5000/admin/2

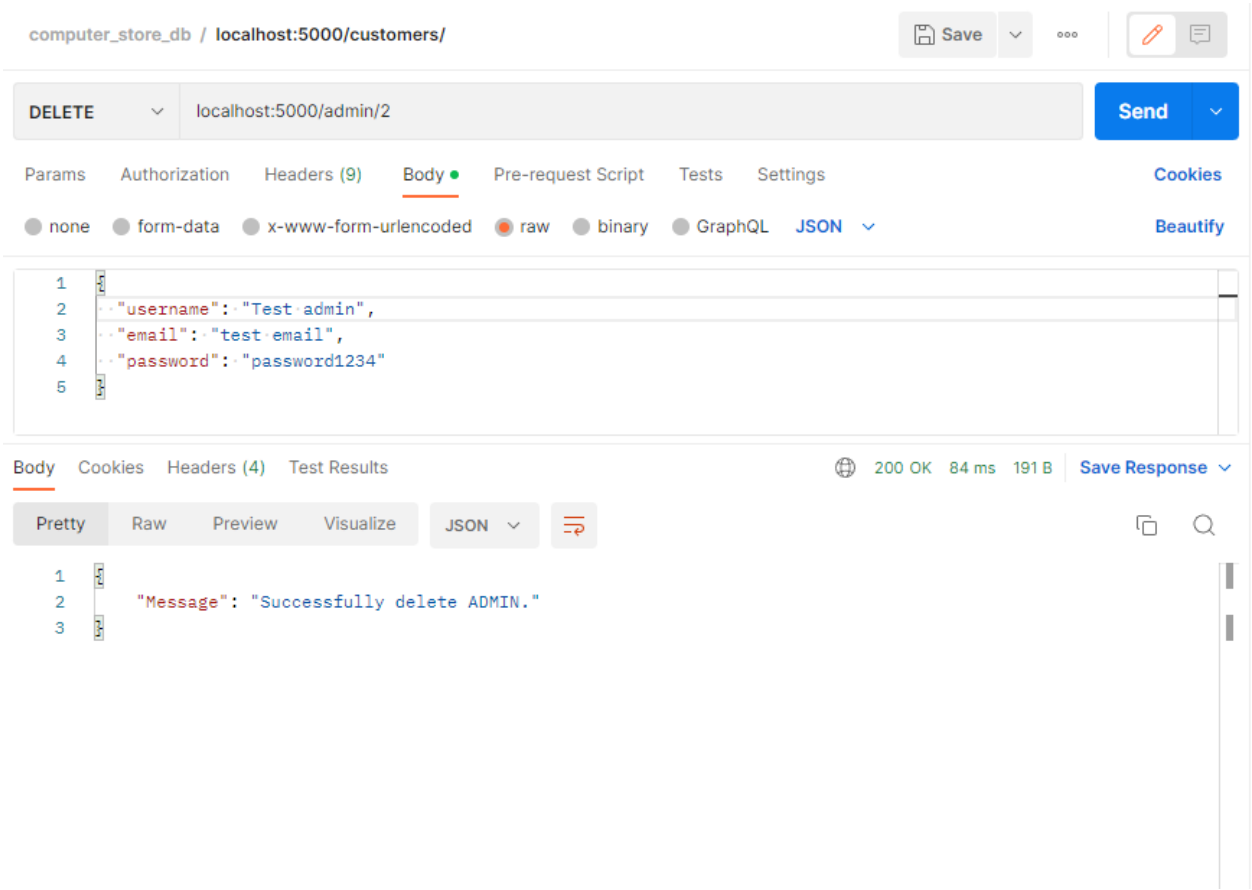
Functionality and Required Fields:

This method takes no fields, it only takes the integer ID of the administrator that is being deleted.

NOTE: To delete an administrator, you need to provide the ID of that administrator at the end of the address. In this example, the admin we created had the ID 2, so to delete that admin, I provided the integer 2 to the end of the address.

Expected return

Successful deletion of an administrator returns a message.



CPU

The CPU controller has 5 end points. None of these end points need a token to function.

Endpoint 1:

Request VERB = GET

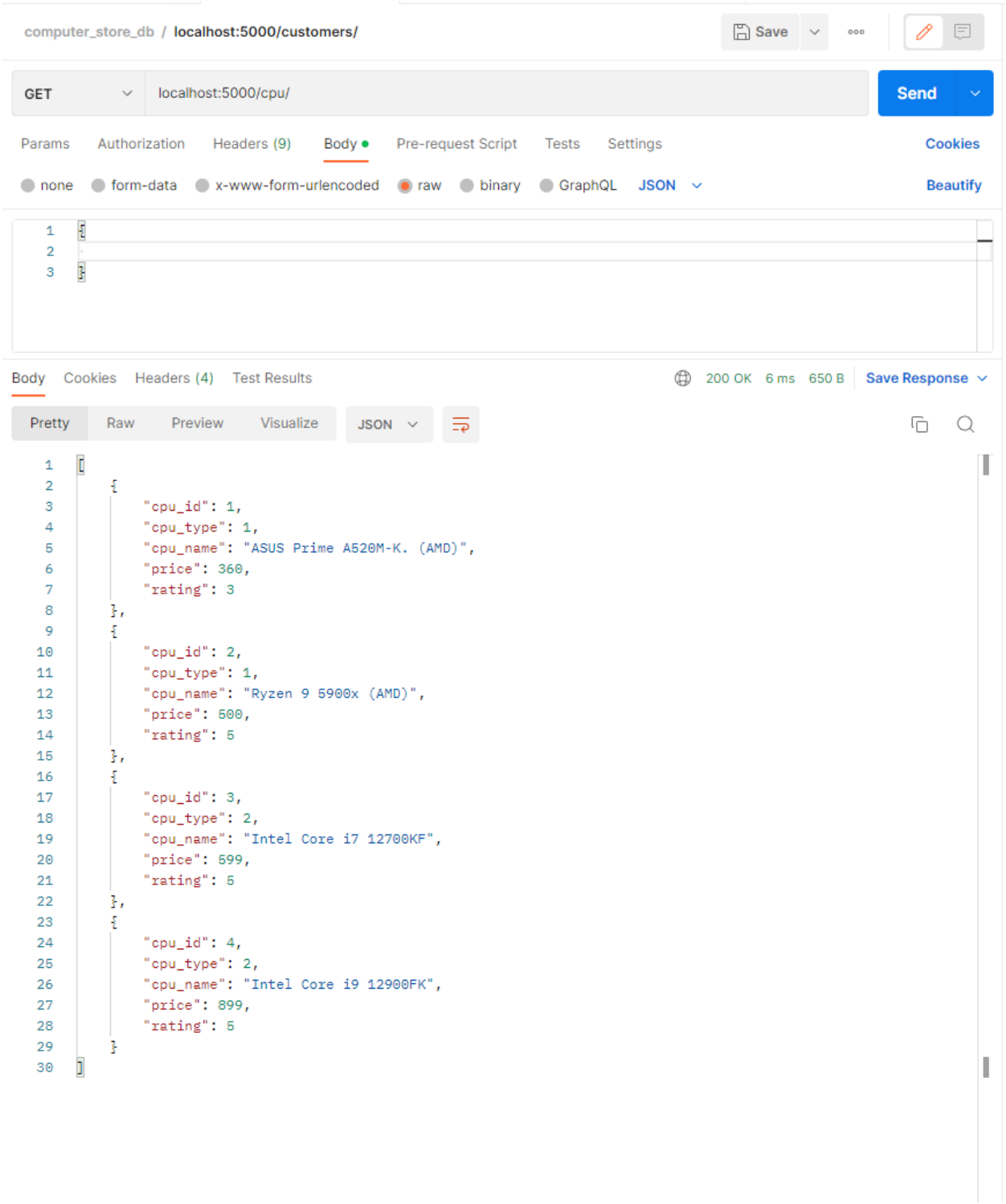
Request ADDRESS = localhost:5000/cpu/

Functionality and Required Fields:

This method takes no fields, the GET request will return a full list of the CPUs recorded in the database.

Expected return

Successful retrieval should return a list of all CPUs



Endpoint 2:

Request VERB = GET + ID

Request ADDRESS = localhost:5000/cpu/3

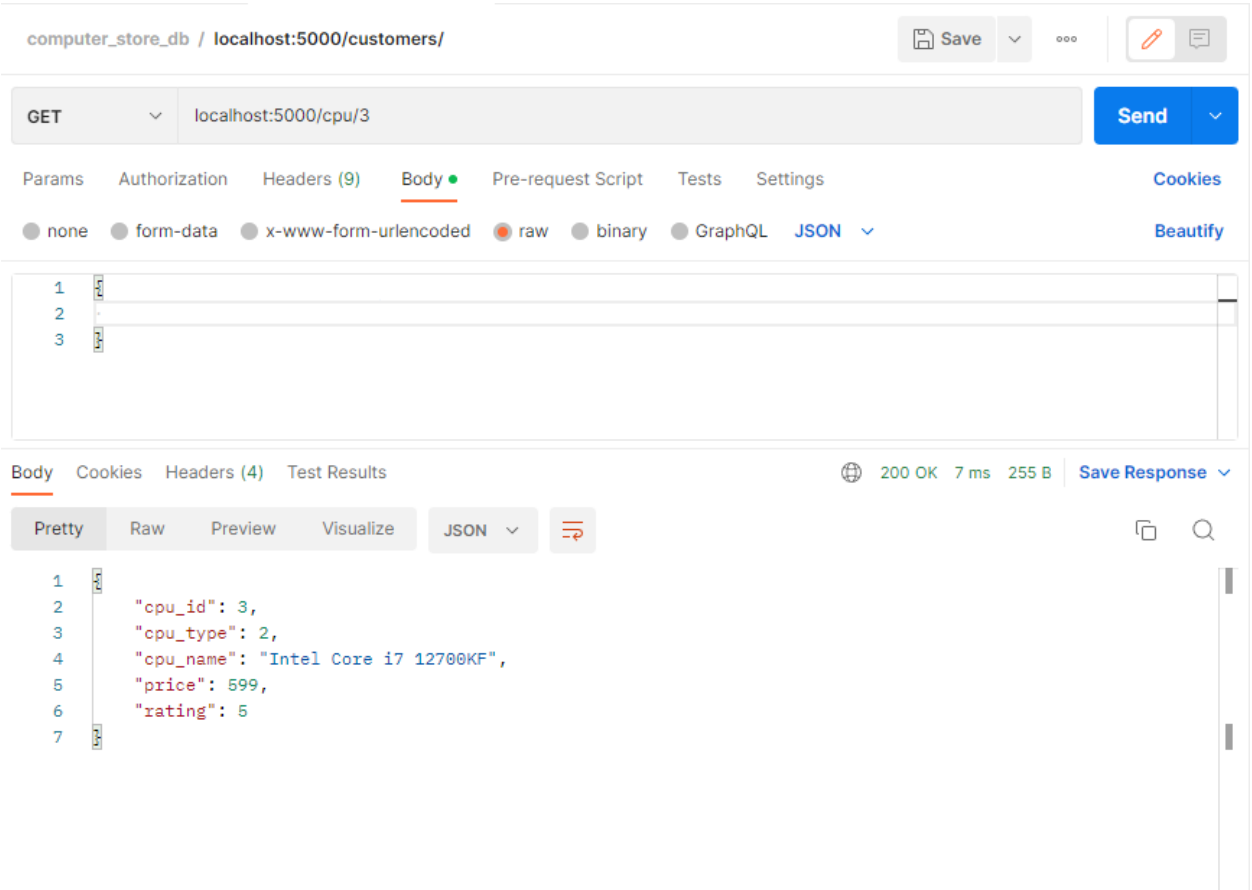
NOTE: The 3 at the end of that link represents the ID of the 3rd CPU. There are 4 CPUs hard coded into the database, to GET one of them, use their respective ID.

Functionality and Required Fields:

This method takes no fields, it only takes the integer ID of the CPU being retrieved.

Expected return

Successful retrieval should return the CPU respective to the ID you entered into the URL.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/cpu/

Functionality and Required Fields:

The required fields and data types to create a CPU are as follows:

cpu_type: Integer

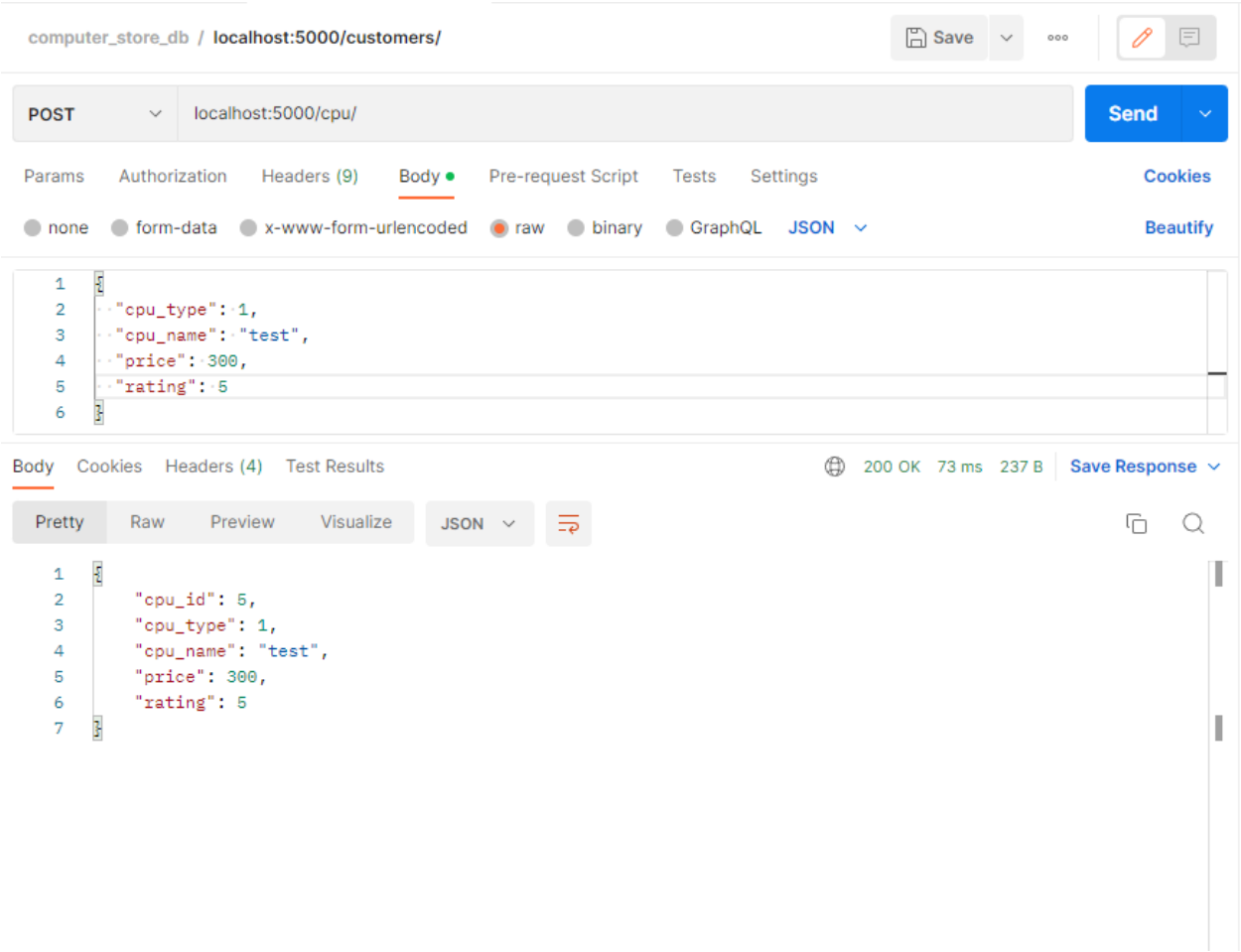
cpu_name: String

price: Integer

rating: Integer

Expected return

Successful creation of CPU will return the information you entered alongside an assigned ID.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/cpu/5

NOTE: The 5 at the end of that link represents the ID of the 5th CPU (the one I'm updating in this example). There are 4 CPUs hard coded into the database, to update one of them, use their respective ID. In this example, I created a CPU and then Updated it.

Functionality and Required Fields:

The required fields and data types to update a CPU are as follows:

cpu_id: must be entered in the URL as an integer

cpu_type: Integer

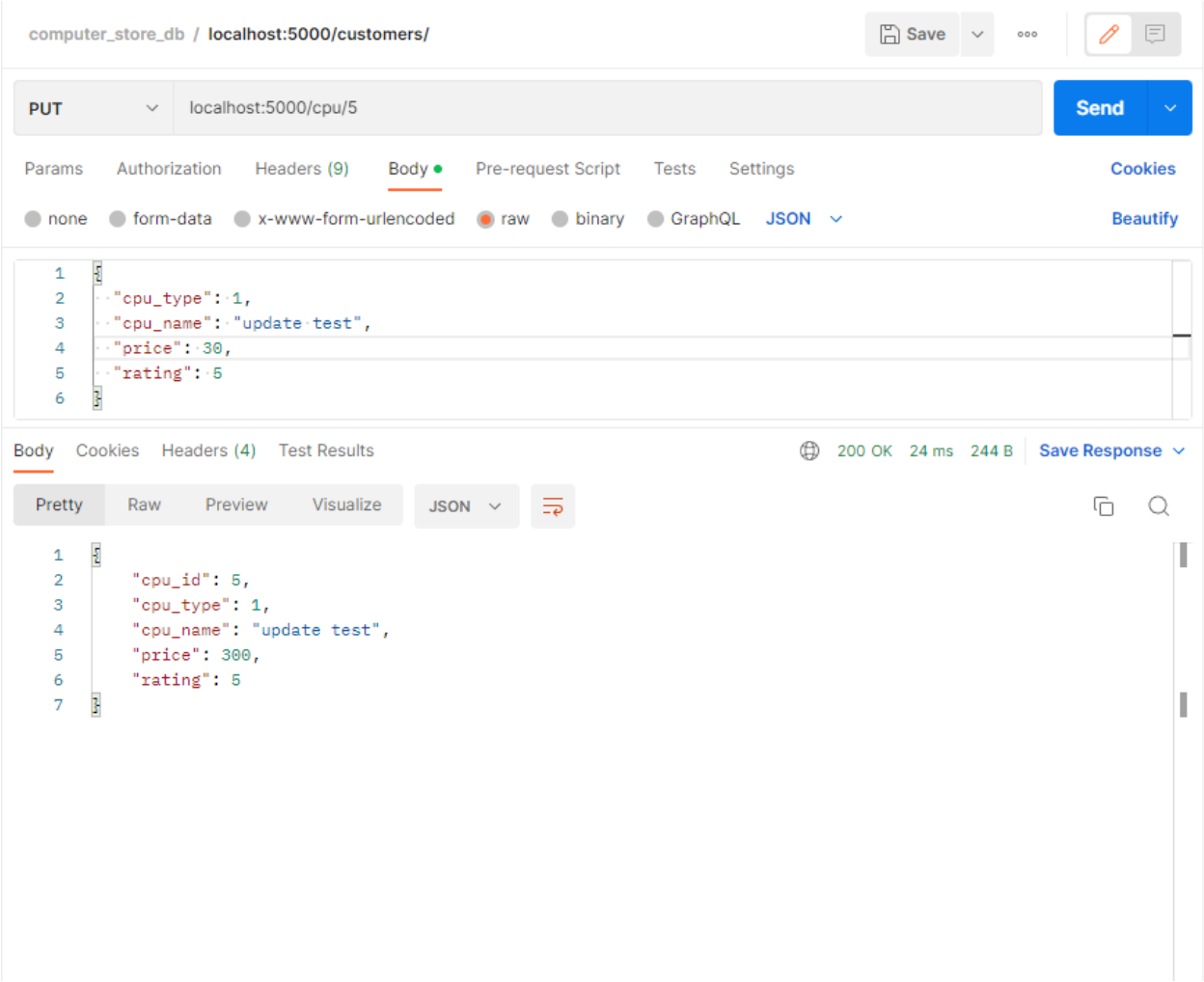
cpu_name: String

price: Integer

rating: Integer

Expected return

Successful update of the cpu will return the updated details to you, alongside the ID associated with it.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/cpu/5

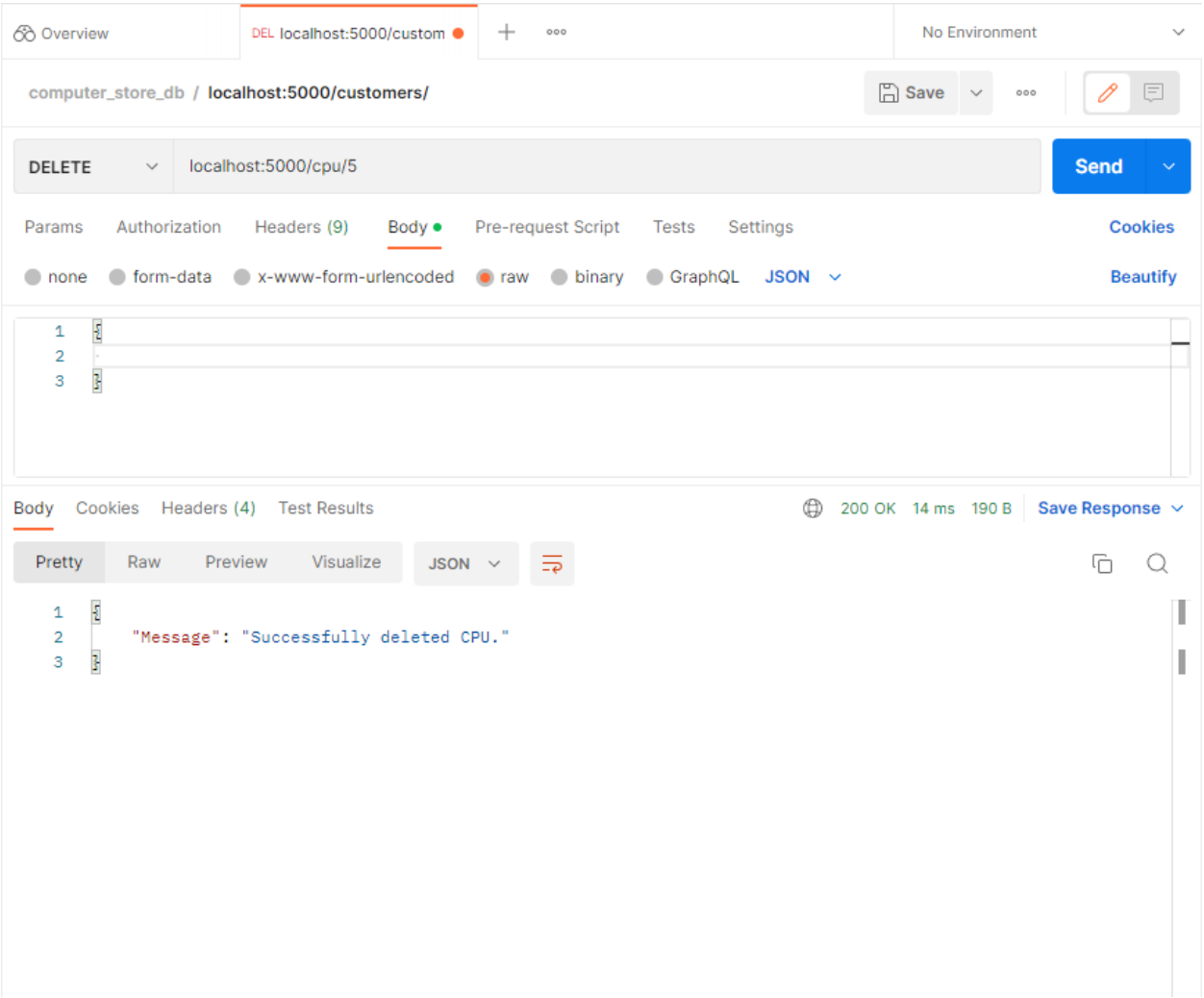
NOTE: The 5 at the end of that link represents the ID of the 5th CPU (the one I'm deleting in this example).

Functionality and Required Fields:

This method takes no fields, it only takes the integer ID of the CPU being deleted.

Expected return

Successful deletion of the cpu will return a message.



CUSTOMERS

The customers controller has 5 end points.

Endpoint 1:

Request VERB = GET

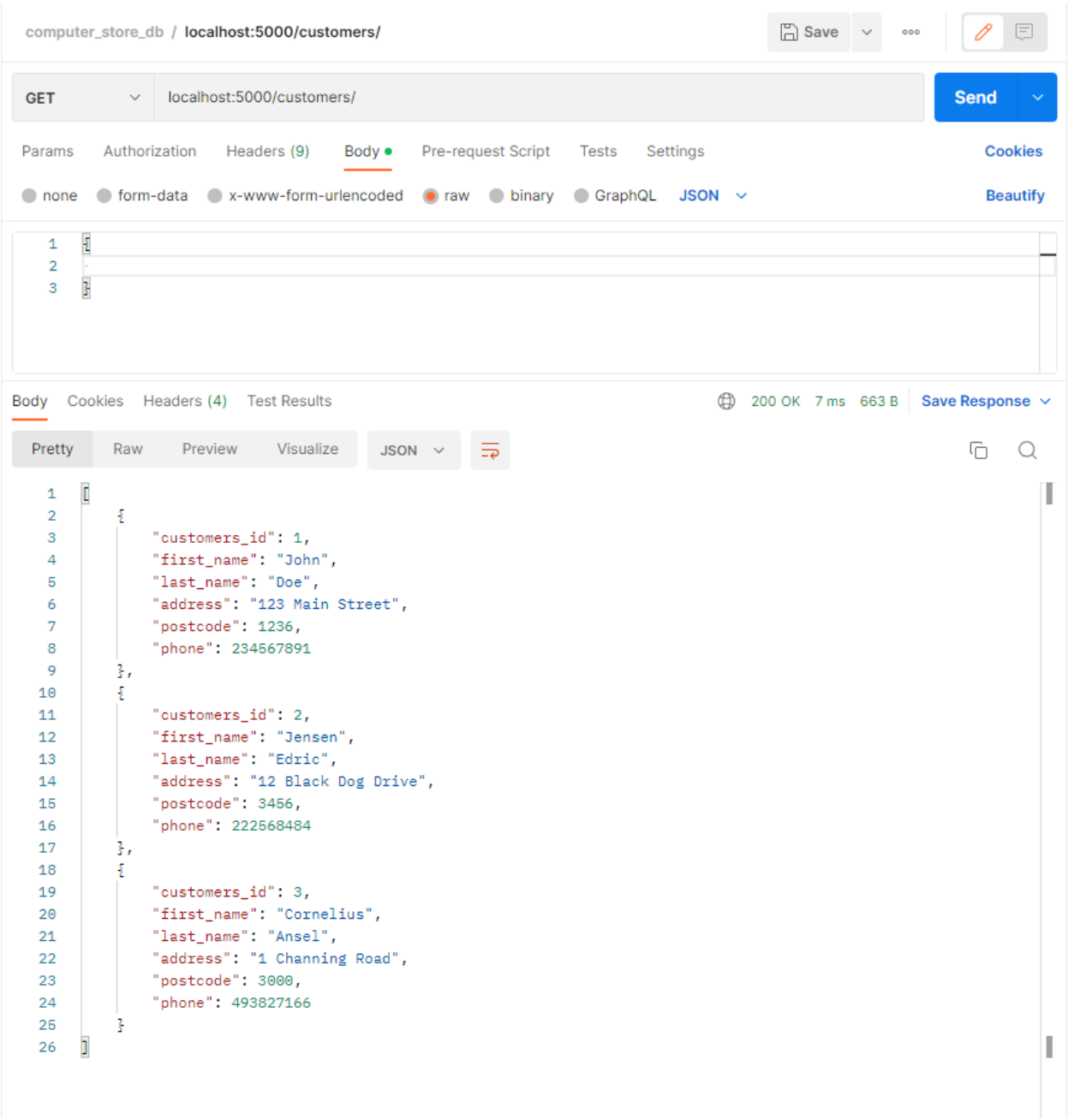
Request ADDRESS = localhost:5000/customers/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all Customers



Endpoint 2:

Request VERB = GET

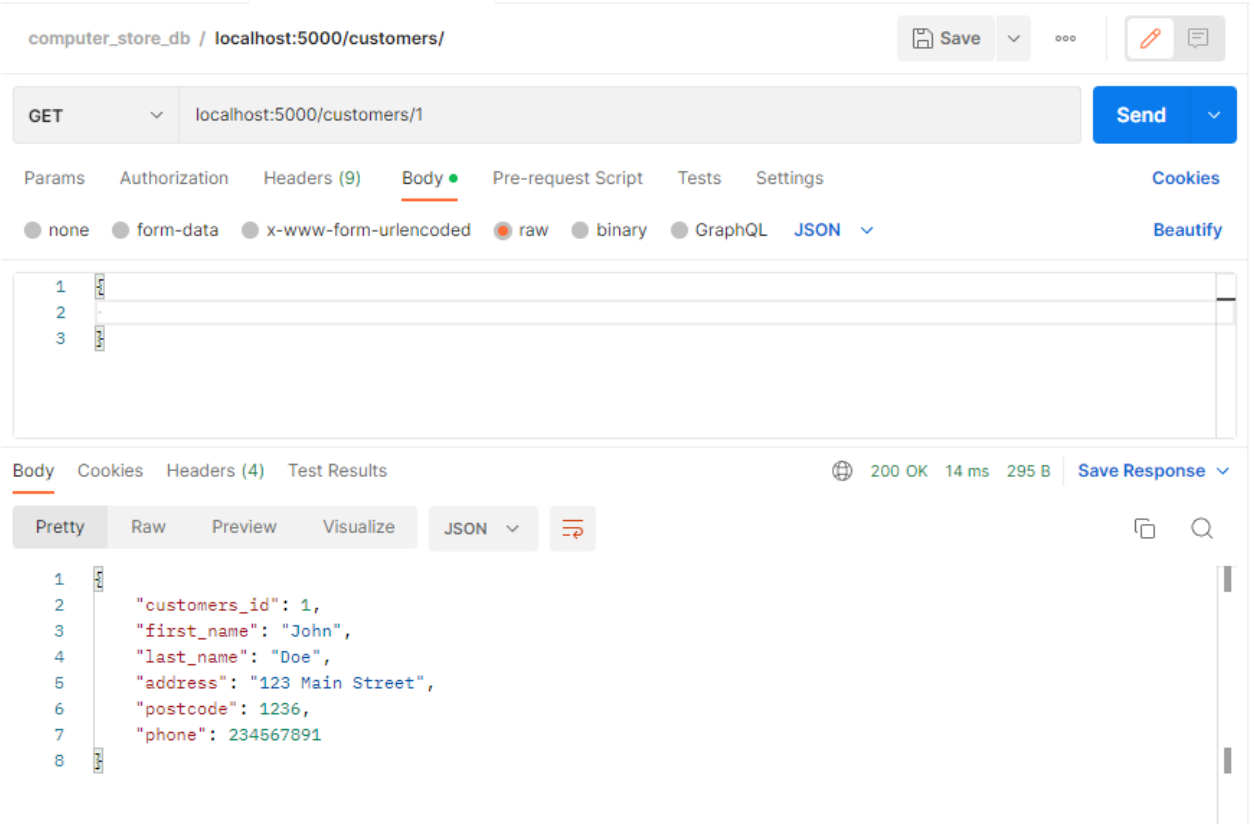
Request ADDRESS = localhost:5000/customers/1

Functionality and Required Fields:

This method takes no fields. It only takes the integer ID of the customer being retrieved at the end of the URL.

Expected return

Successful retrieval should return a list of the Customer associated with the ID you entered.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/customers/

Functionality and Required Fields:

The required fields and data types to create a Customer are as follows:

first_name: String

last_name: String

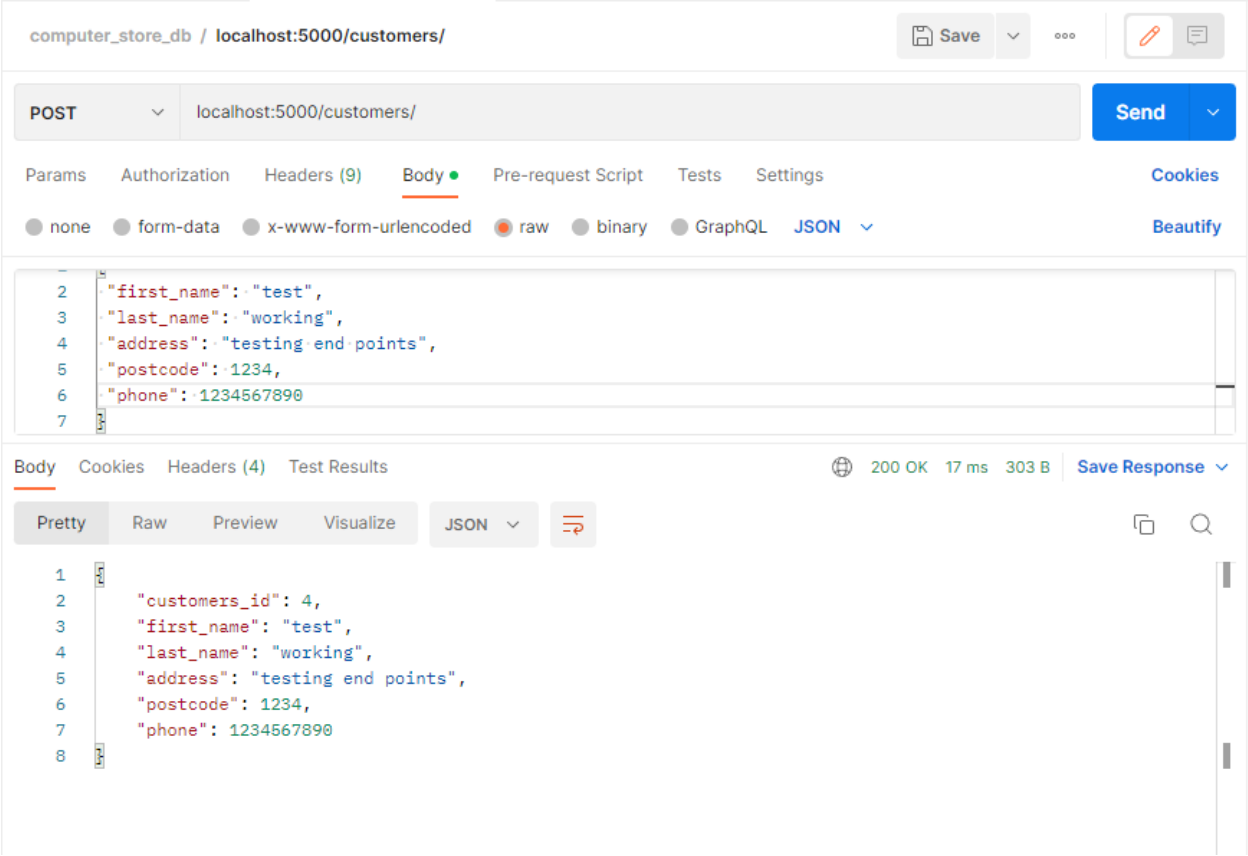
address: String

postcode: Integer

phone: Integer

Expected return

Successful creation of a Customer will return the information you entered alongside an assigned ID.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/customers/4

NOTE: The 4 at the end of that link represents the ID of the 4th Customer (the one I'm updating in this example).

Functionality and Required Fields:

The required fields and data types to update a Customer are as follows:

customer_id: must be entered in the URL as an integer

first_name: String

last_name: String

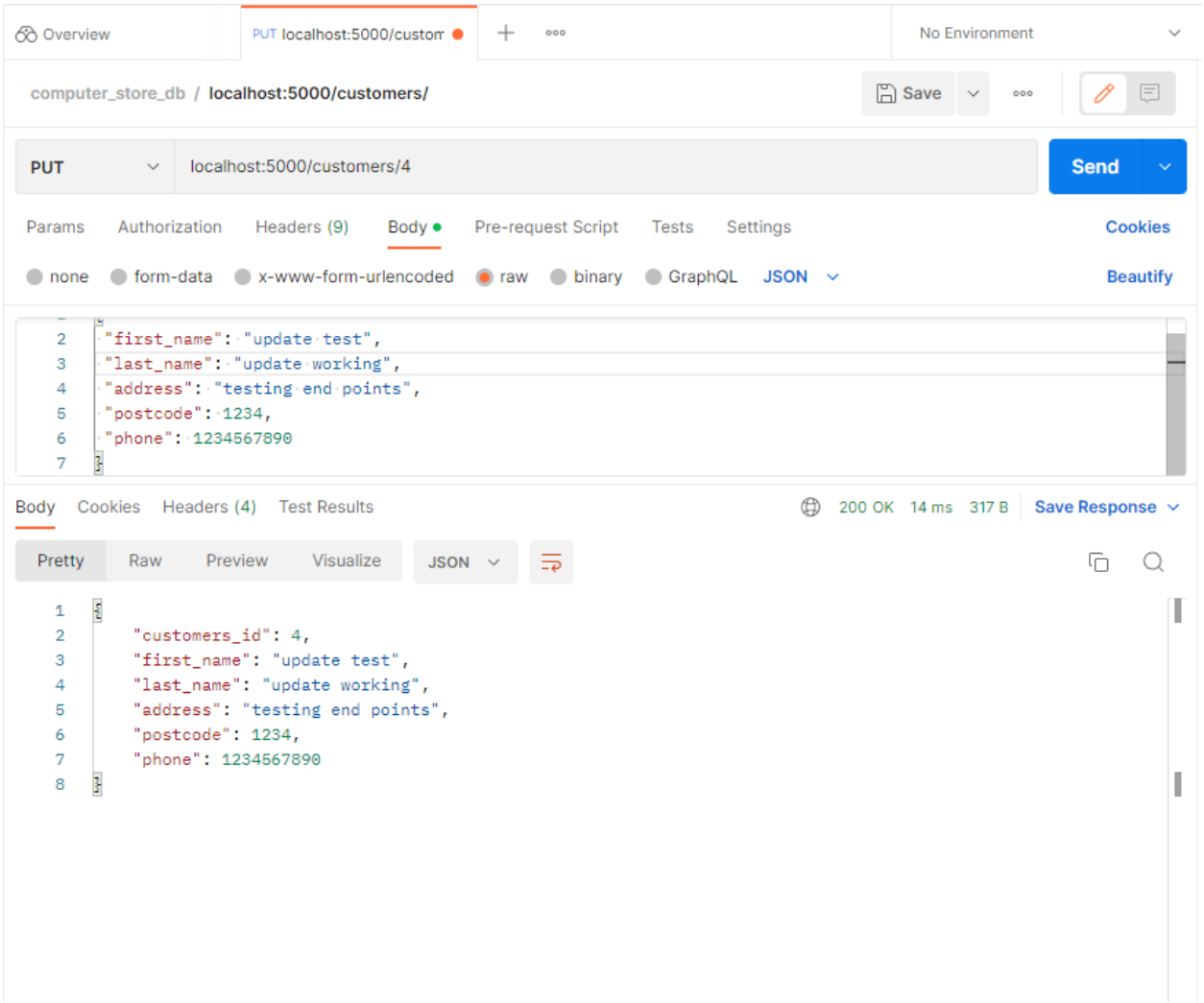
address: String

postcode: Integer

phone: Integer

Expected return

Successful update of a Customer will return the information you entered alongside an assigned ID.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/customers/4

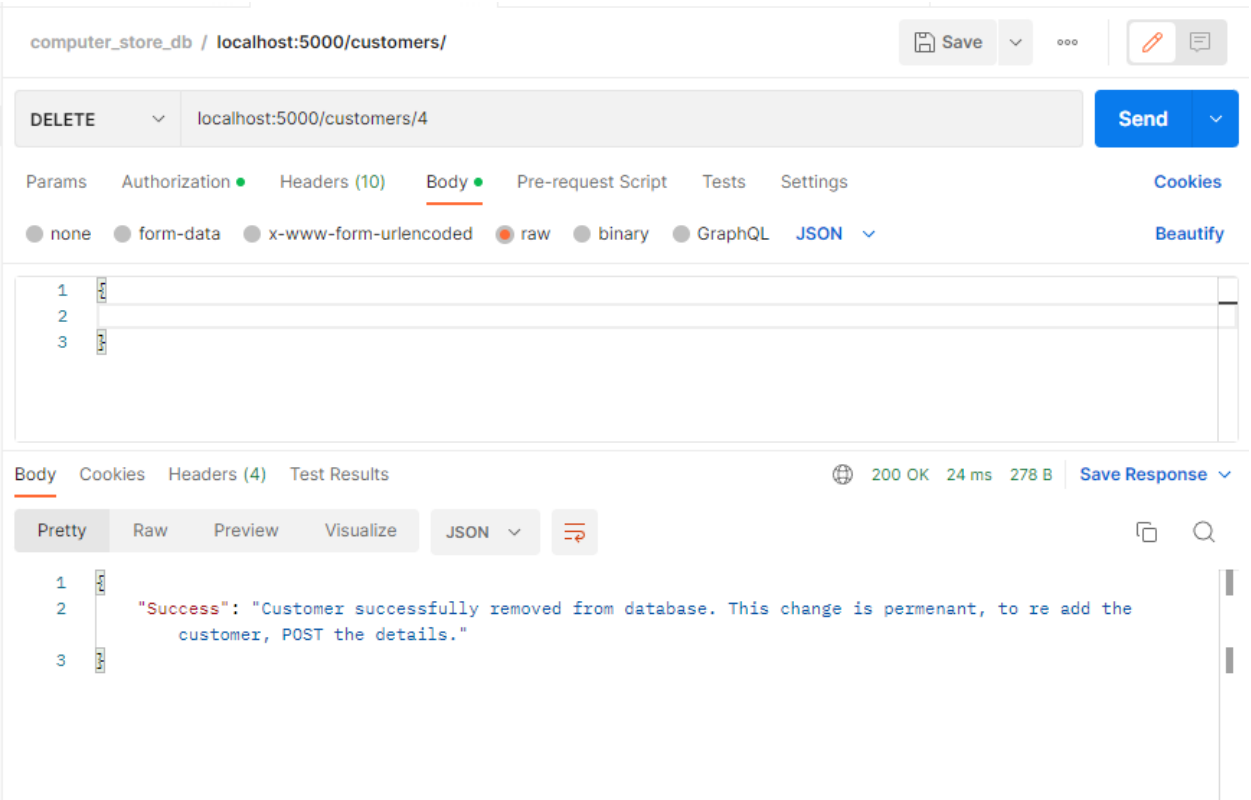
NOTE: The 4 at the end of that link represents the ID of the 4th Customer (the one I'm deleting in this example).

Functionality and Required Fields:

This method takes no fields, it only takes the integer ID of the Customer being deleted. Deleting a customer is however protected by JWT, an administrator token is needed to delete a customer.

Expected return

Successful deletion of the customer will return a message.



GPU

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

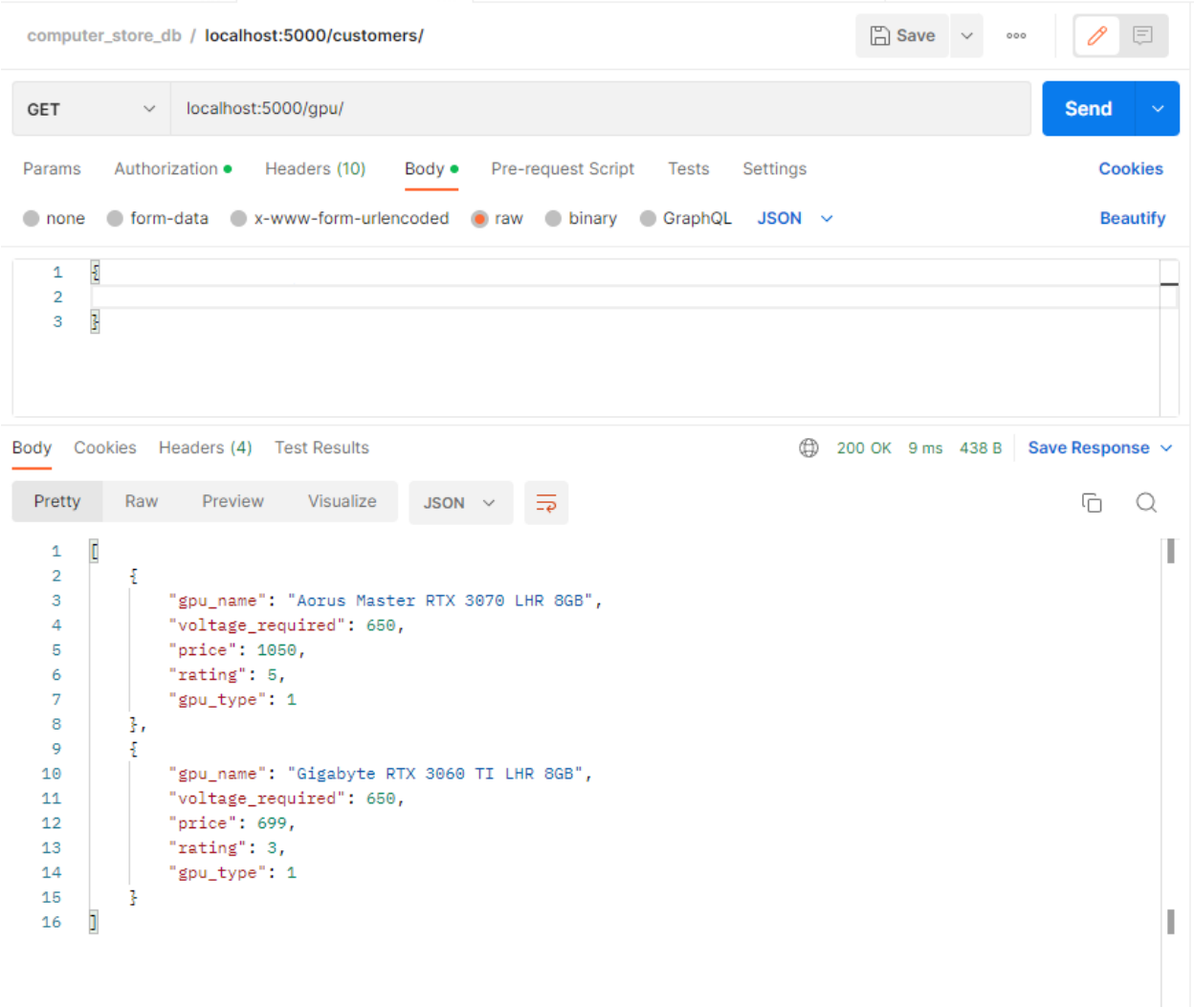
Request ADDRESS = localhost:5000/gpu/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all GPUs.



Endpoint 2:

Request VERB = GET

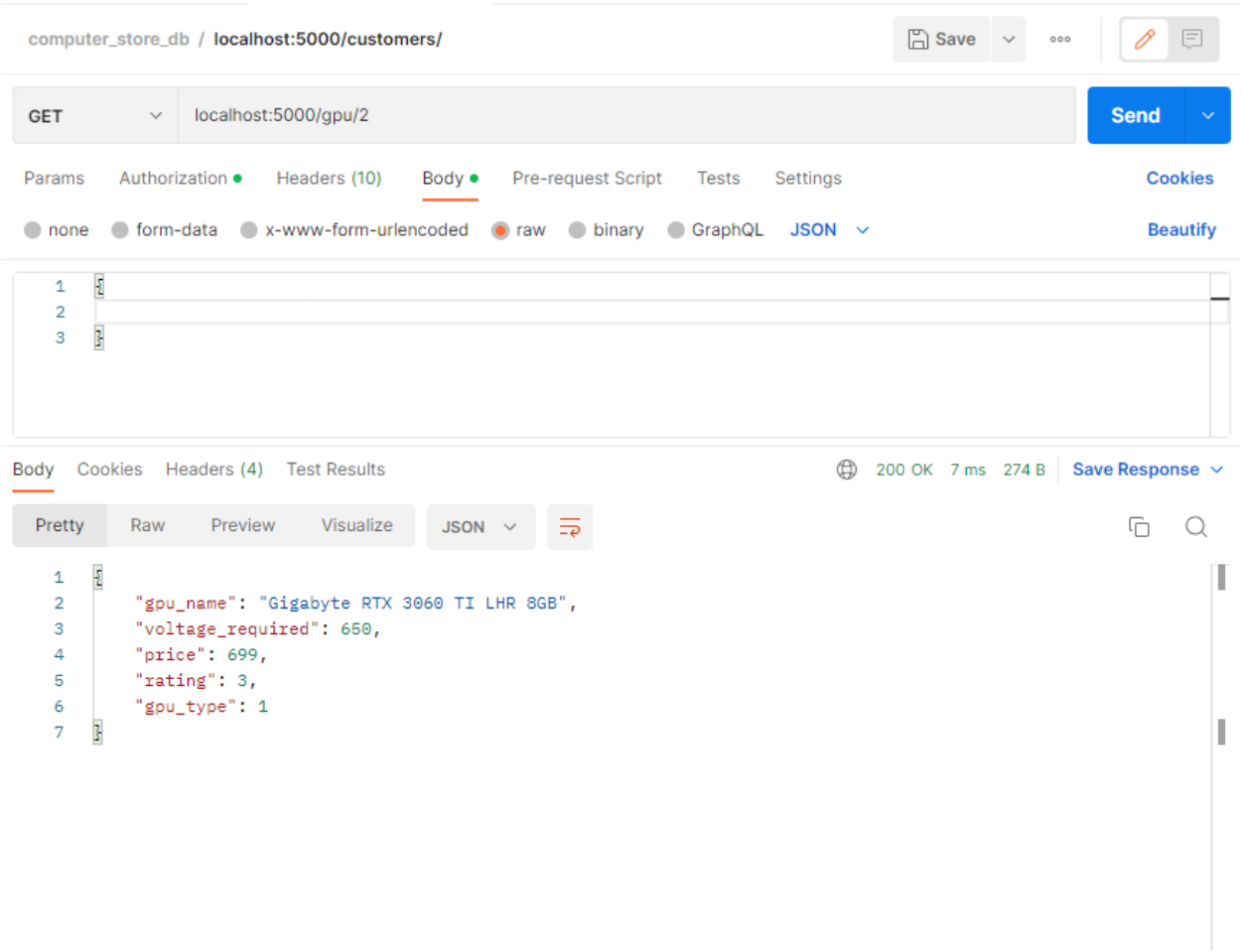
Request ADDRESS = localhost:5000/gpu/2

Functionality and Required Fields:

This method takes no fields. It only takes the integer ID of the GPU being retrieved at the end of the URL

Expected return

Successful retrieval should return the GPU associated with the ID you entered.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/gpu/

Functionality and Required Fields:

The required fields and data types to create a Customer are as follows:

gpu_name: String

voltage_required: Integer

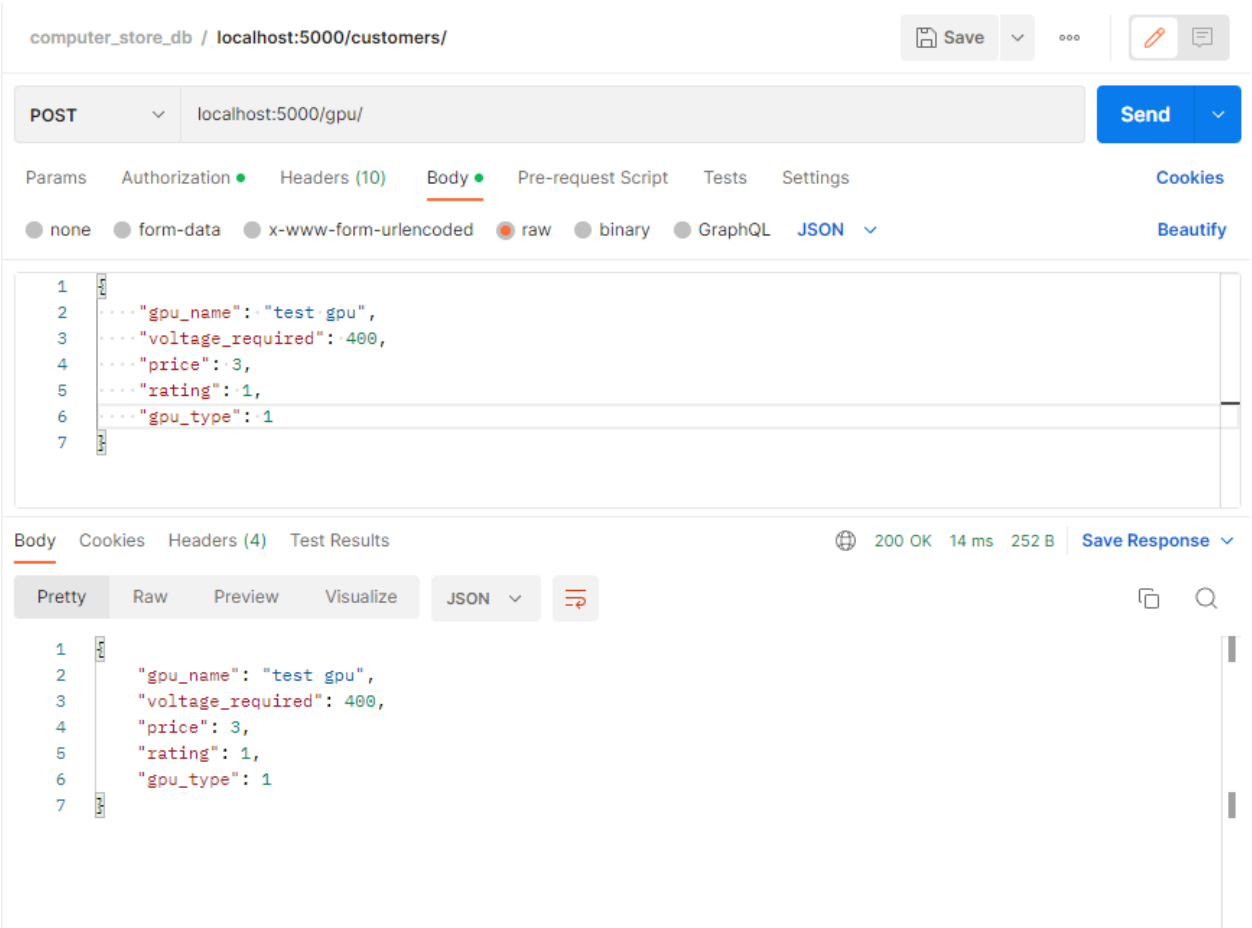
price: Integer

rating: Integer

gpu_type: Integer

Expected return

Successful creation of a GPU will return the information you entered. It does not return an ID; I was experimenting with fields in the schema. This is intentional.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/gpu/3

NOTE: The 3 at the end of that link represents the ID of the 3rd GPU. There are 2 GPUs hard coded into the database, to update (PUT) one of them, use their respective ID.

Functionality and Required Fields:

The required fields and data types to create a Customer are as follows:

gpu_name: String

voltage_required: Integer

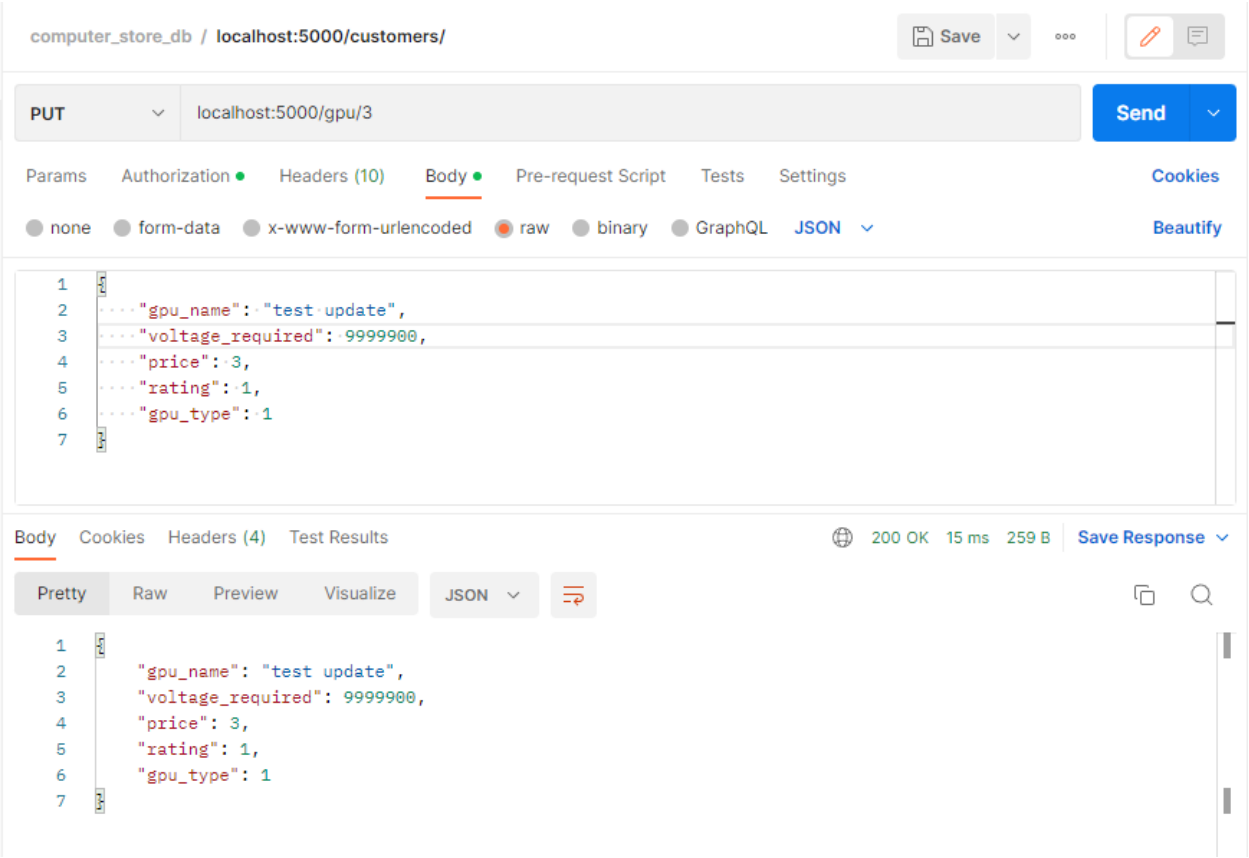
price: Integer

rating: Integer

gpu_type: Integer

Expected return

Successful creation of a GPU will return the information you entered. It does not return an ID; I was experimenting with fields in the schema. This is intentional.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/gpu/3

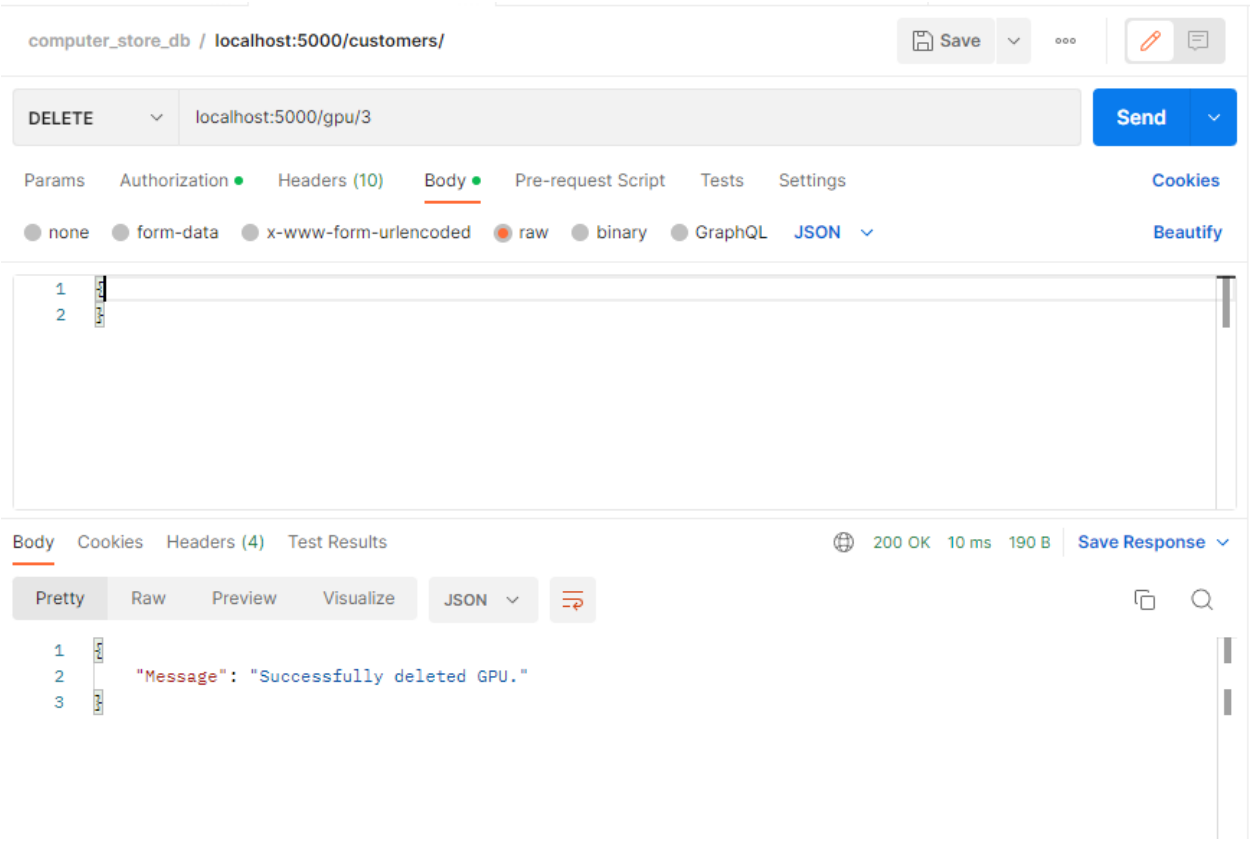
NOTE: The 3 at the end of that link represents the ID of the 3rd GPU (the one I'm deleting in this example).

Functionality and Required Fields:

This method takes no fields, it only takes the integer ID of the GPU being deleted.

Expected return

Successful deletion of the GPU will return a message.



Motherboard and CPU Compatibility (mobo_cpu_compat)

This controller has 1 end point.

Endpoint 1:

Request VERB = GET

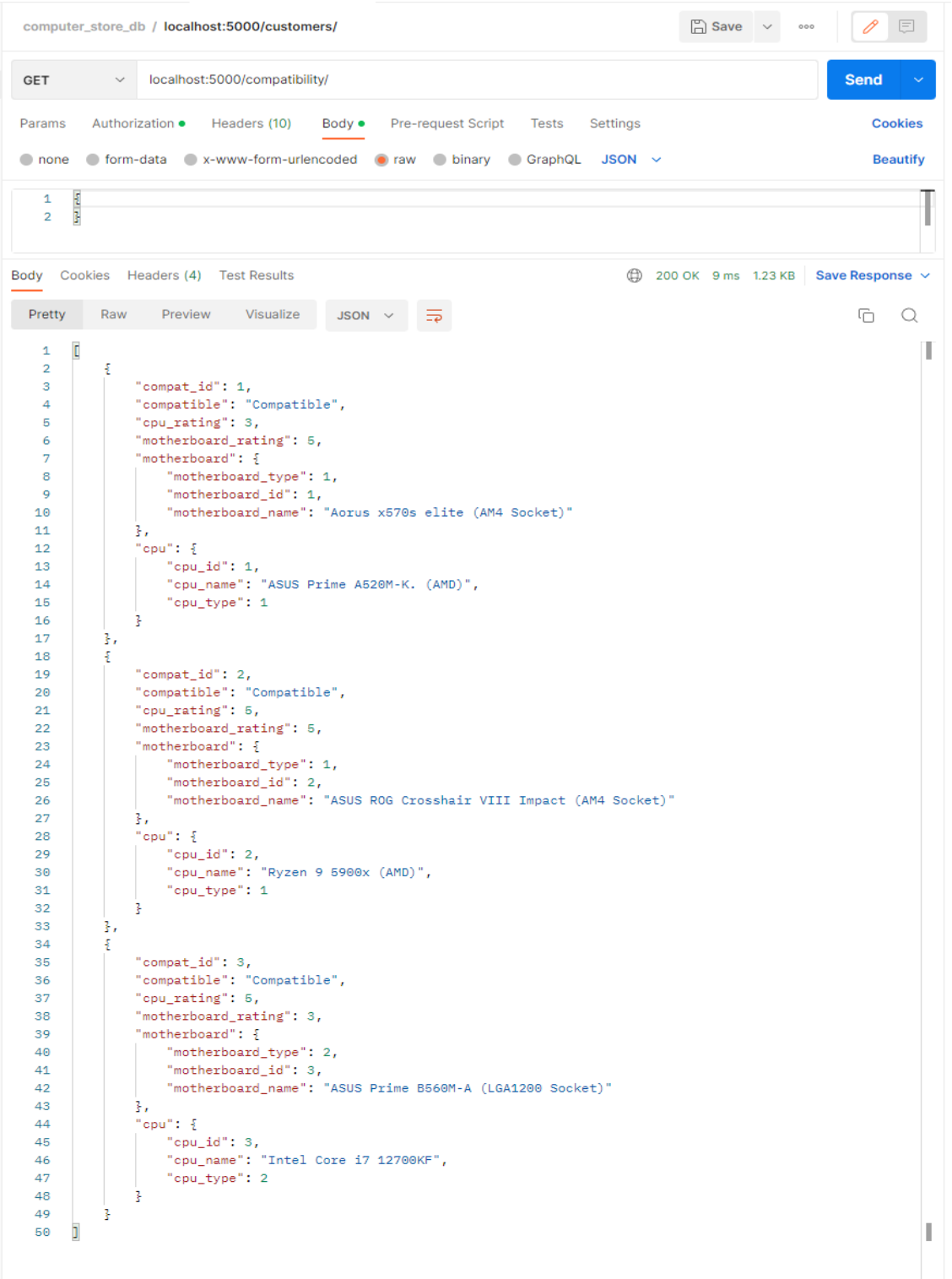
Request ADDRESS = localhost:5000/compatibility/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the compatibility comparisons. Upon retrieval of the list, there should be 2 nested fields (motherboard and cpu), these 2 fields are nested withing the compatibility fields.



Motherboards

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

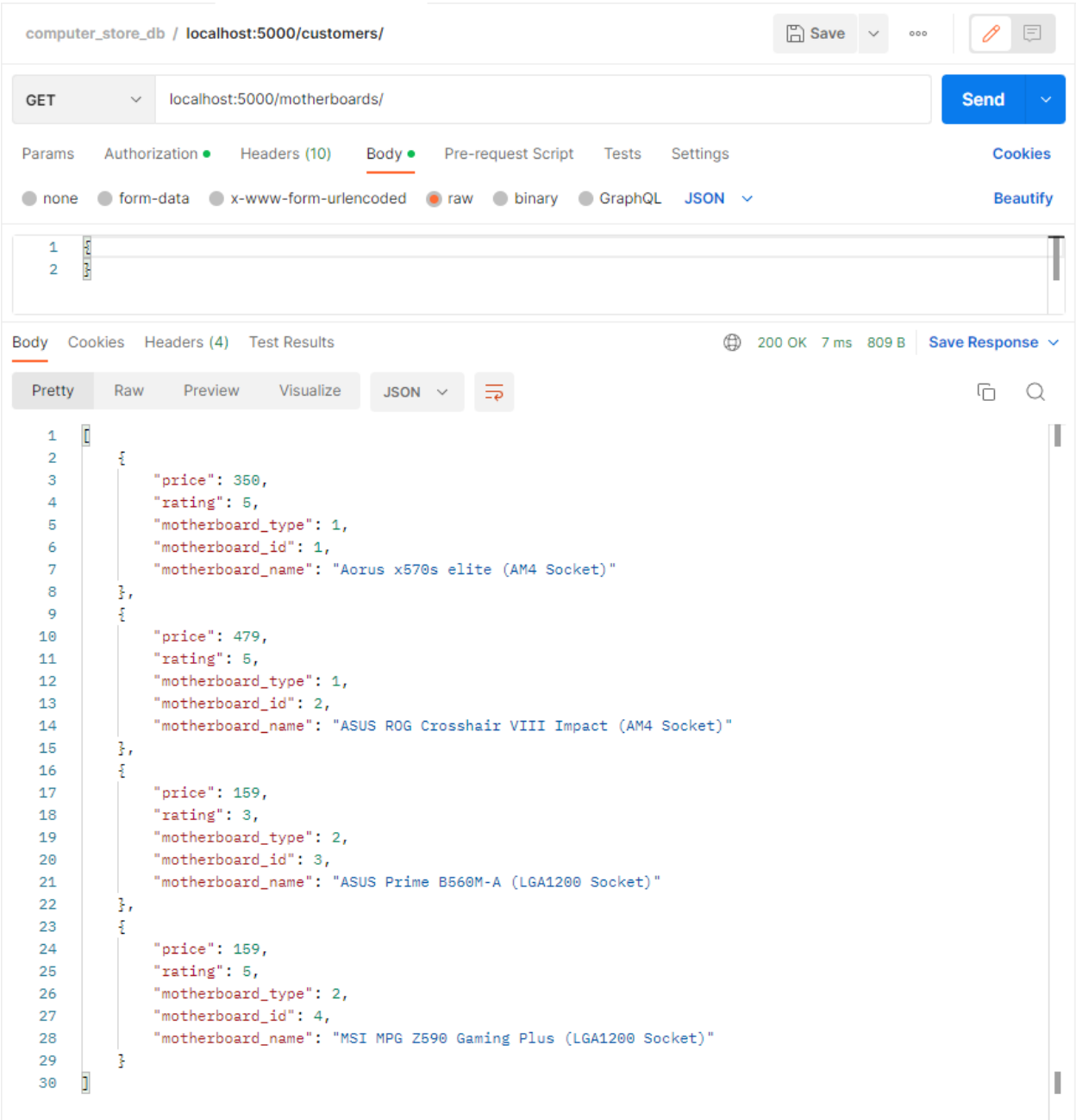
Request ADDRESS = localhost:5000/motherboards/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the motherboards.



Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/motherboards/2

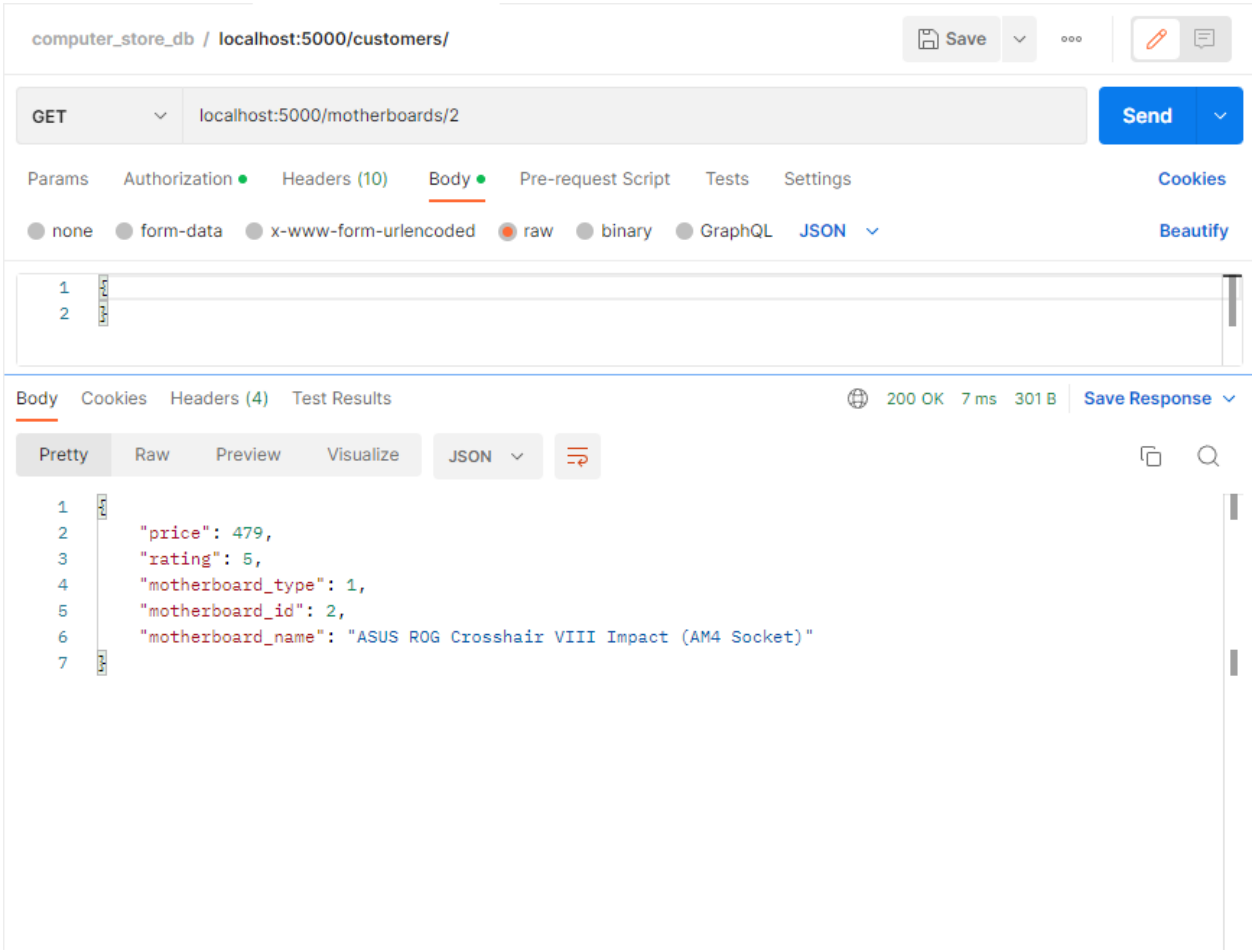
NOTE: The 2 at the end of that link represents the ID of the 2nd motherboard.

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return the motherboard associated with the ID you entered at the end of the URL.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/motherboards/

NOTE: The 2 at the end of that link represents the ID of the 2nd motherboard.

Functionality and Required Fields:

price: Integer

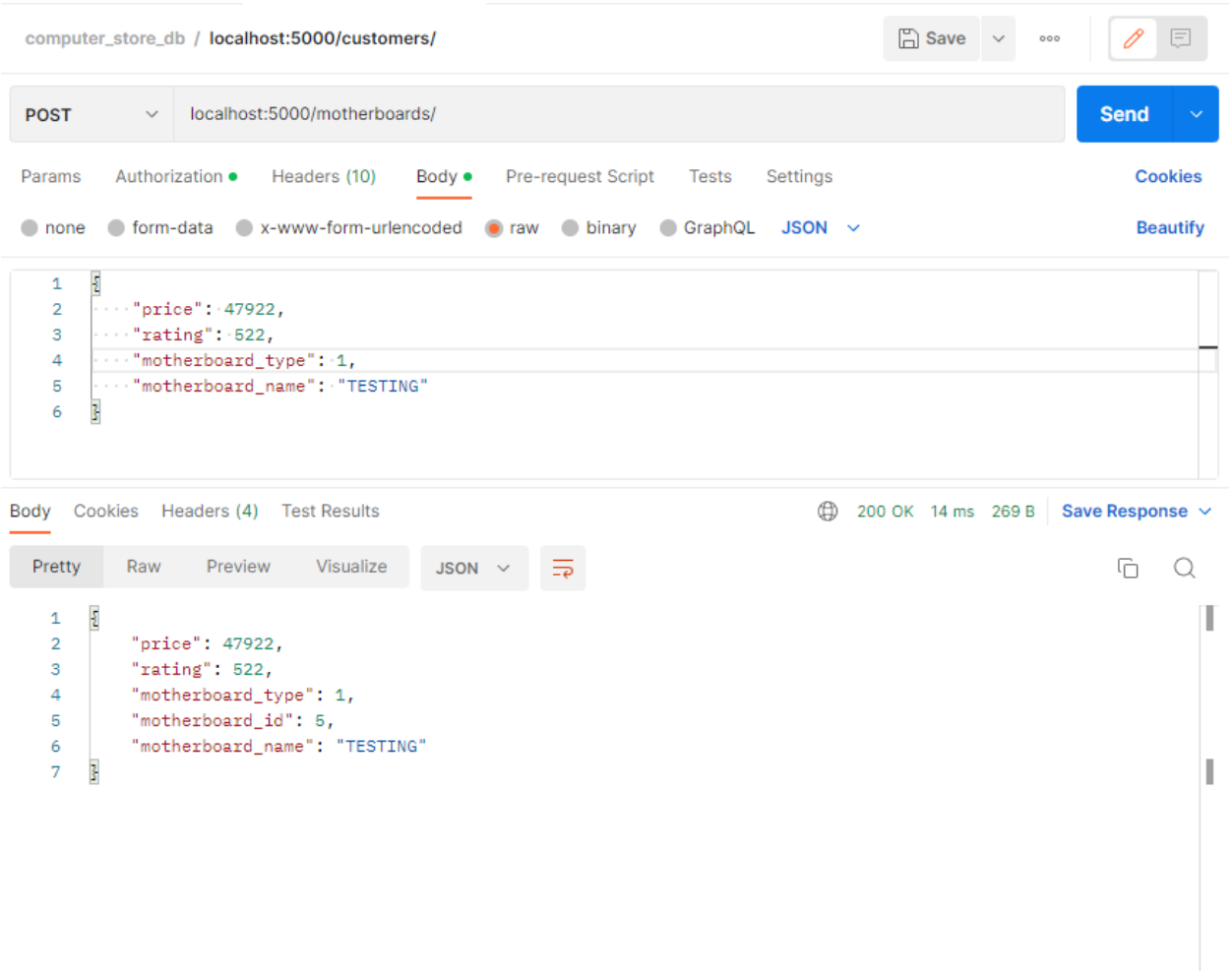
rating: Integer

motherboard_type: Integer

motherboard_name: String

Expected return

Successful creation should return the information you entered alongside an ID.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/motherboards/5

NOTE: The 5 at the end of that link represents the ID of the 5th motherboard. (The one we just created.)

Functionality and Required Fields:

price: Integer

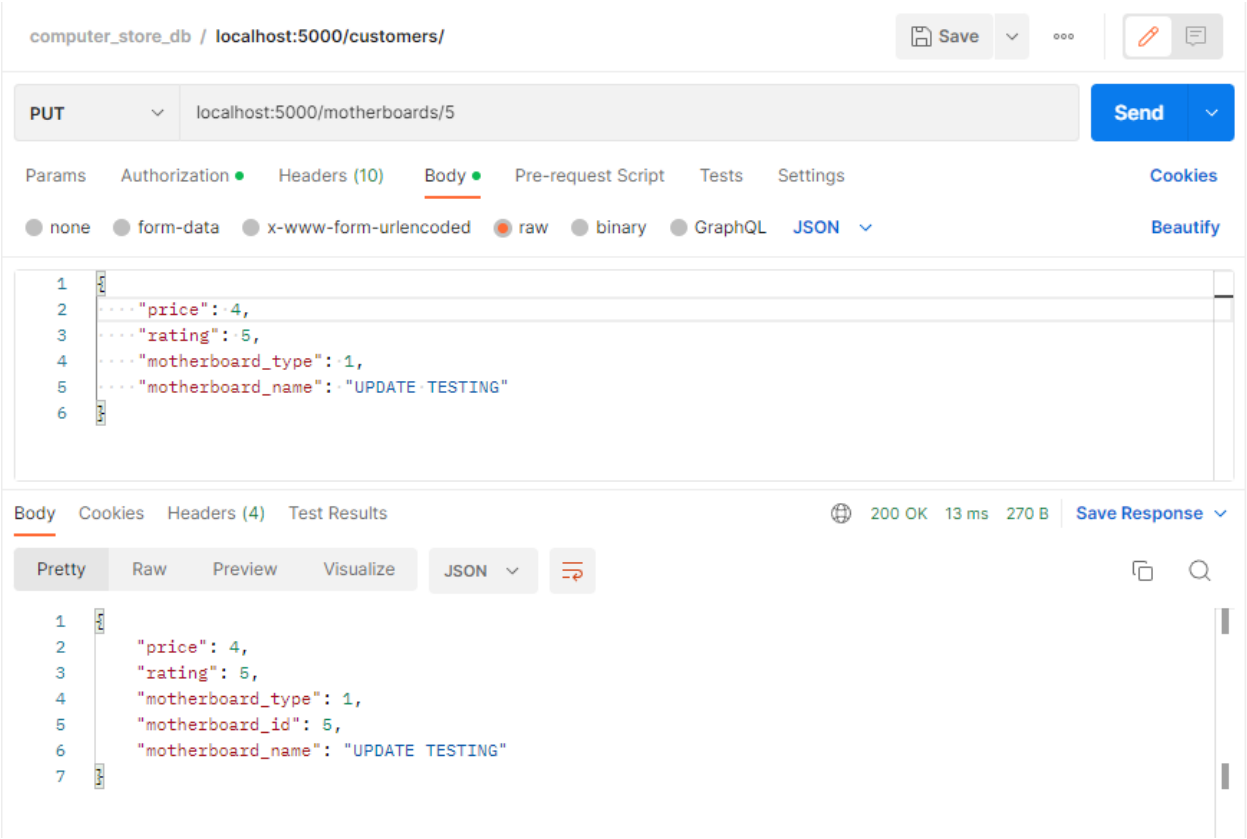
rating: Integer

motherboard_type: Integer

motherboard_name: String

Expected return

Successful creation should return the information you entered alongside an ID.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/motherboards/5

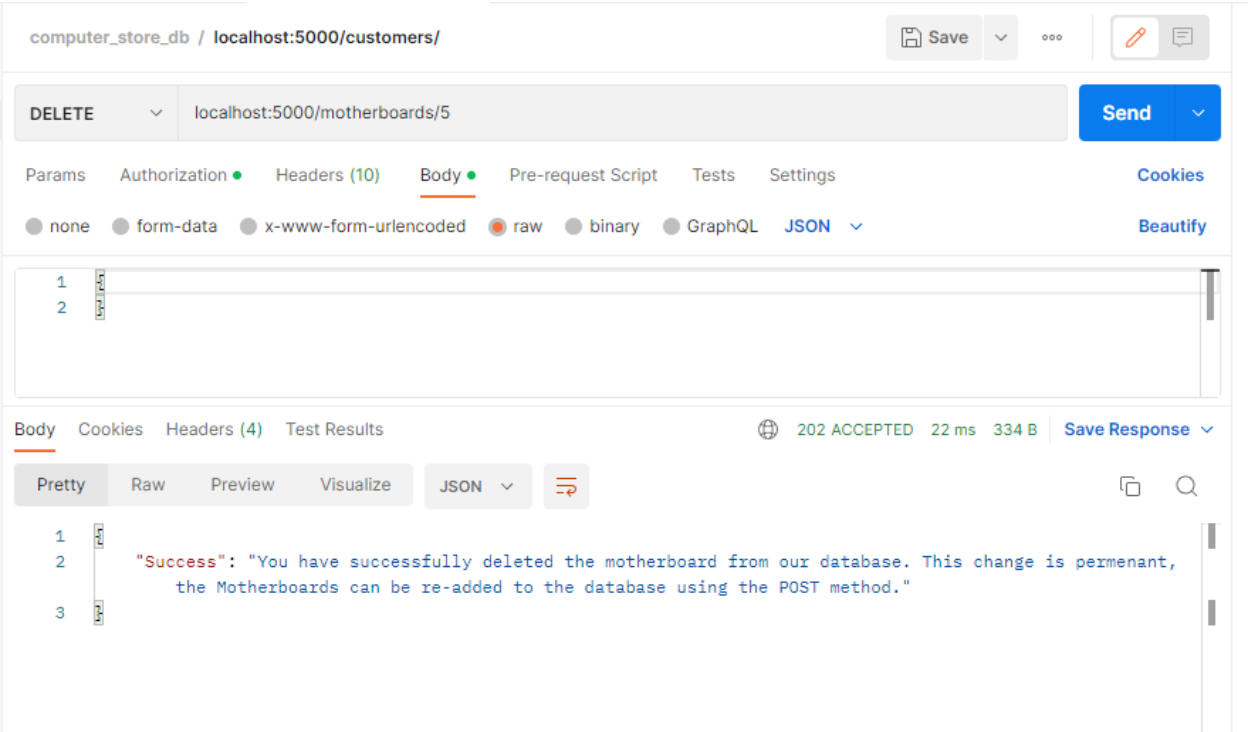
NOTE: The 5 at the end of that link represents the ID of the 5th motherboard. (The one we created, updated and will now delete)

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful deletion of the Motherboard should return a message.



Order

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

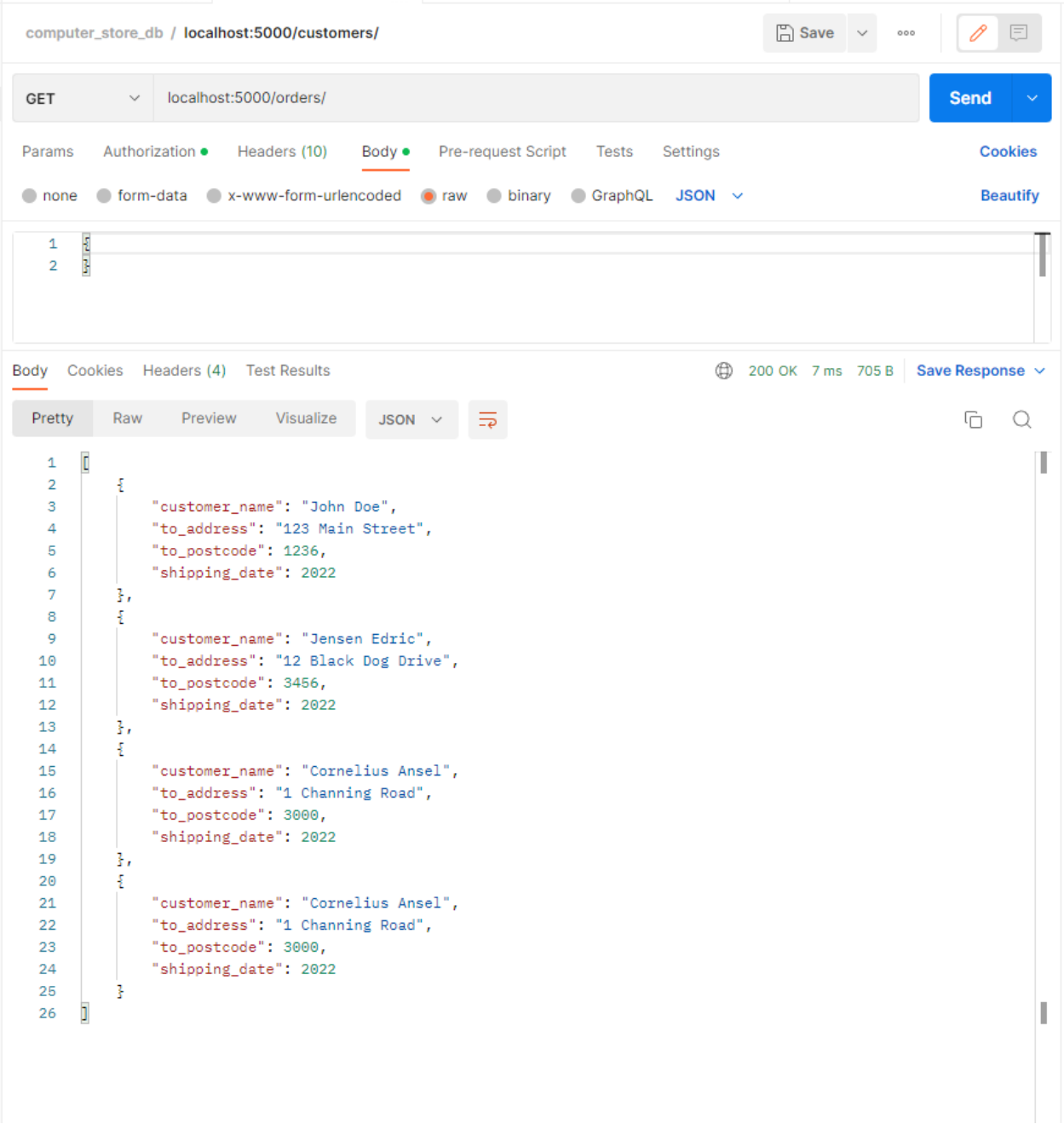
Request ADDRESS = localhost:5000/orders/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the orders.



Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/orders/3

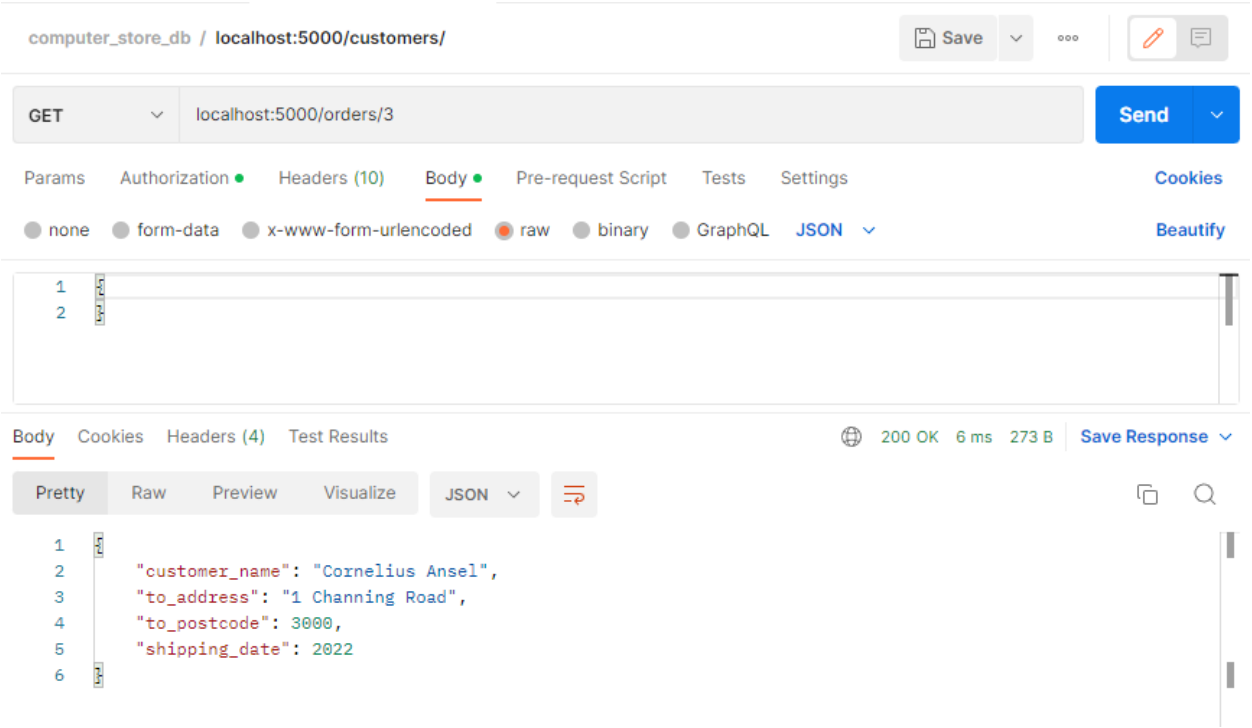
NOTE: The 3 at the end of that link represents the ID of the 3rd Order.

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of the order associated with the ID you entered at the end of the URL.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/orders/

Functionality and Required Fields:

customer_name: String

to_address: String

to_postcode: Integer

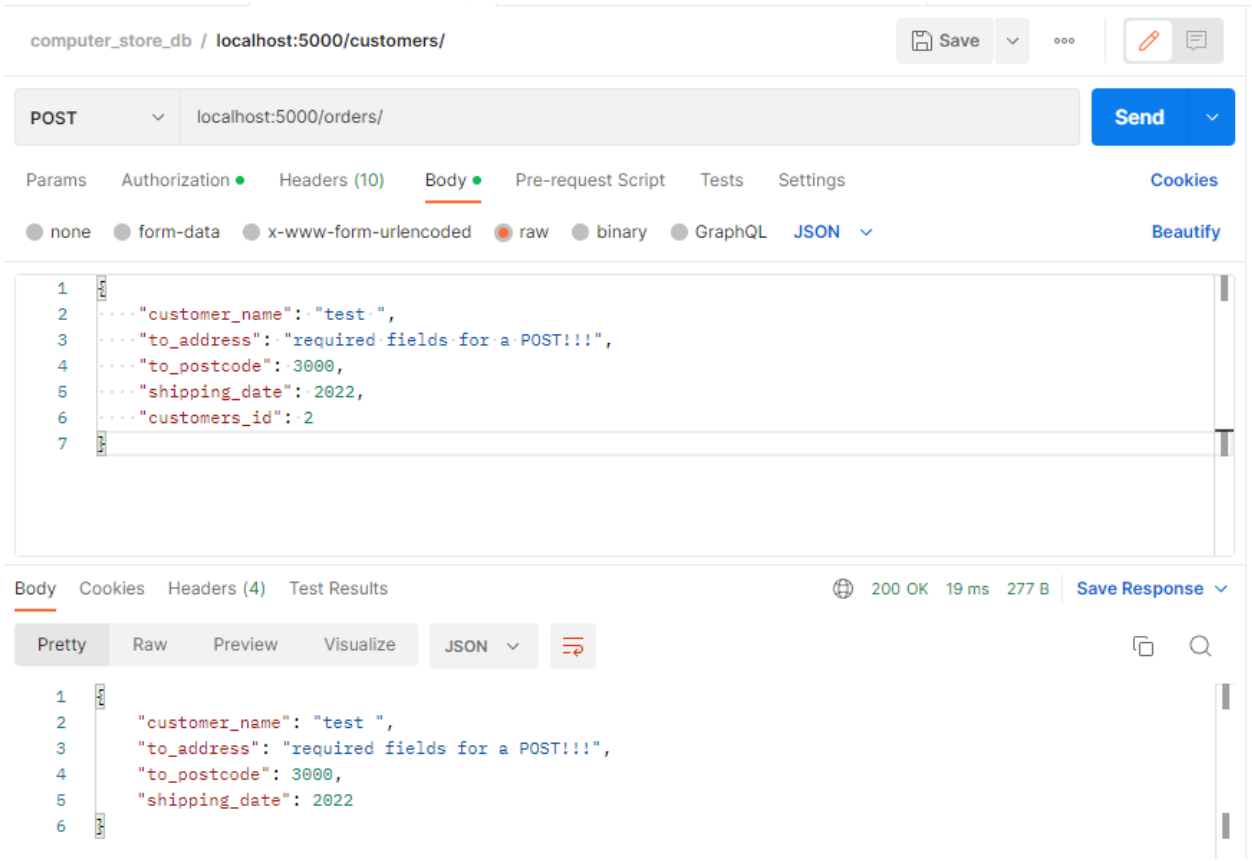
shipping_date: Integer

customers_id: Integer

Expected return

Successful creation should return the information of the order you entered. It will not return the ID.

This method catches all missing fields except for customer_id. If you leave a field out, it will display a message to let you know what's missing. If you leave the ID out, it will throw the default error codes. Didn't have time to correct it.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/orders/5

NOTE: The 5 at the end of that link represents the ID of the 5th order. (The one we created and updated.)

Functionality and Required Fields:

customer_name: String

to_address: String

to_postcode: Integer

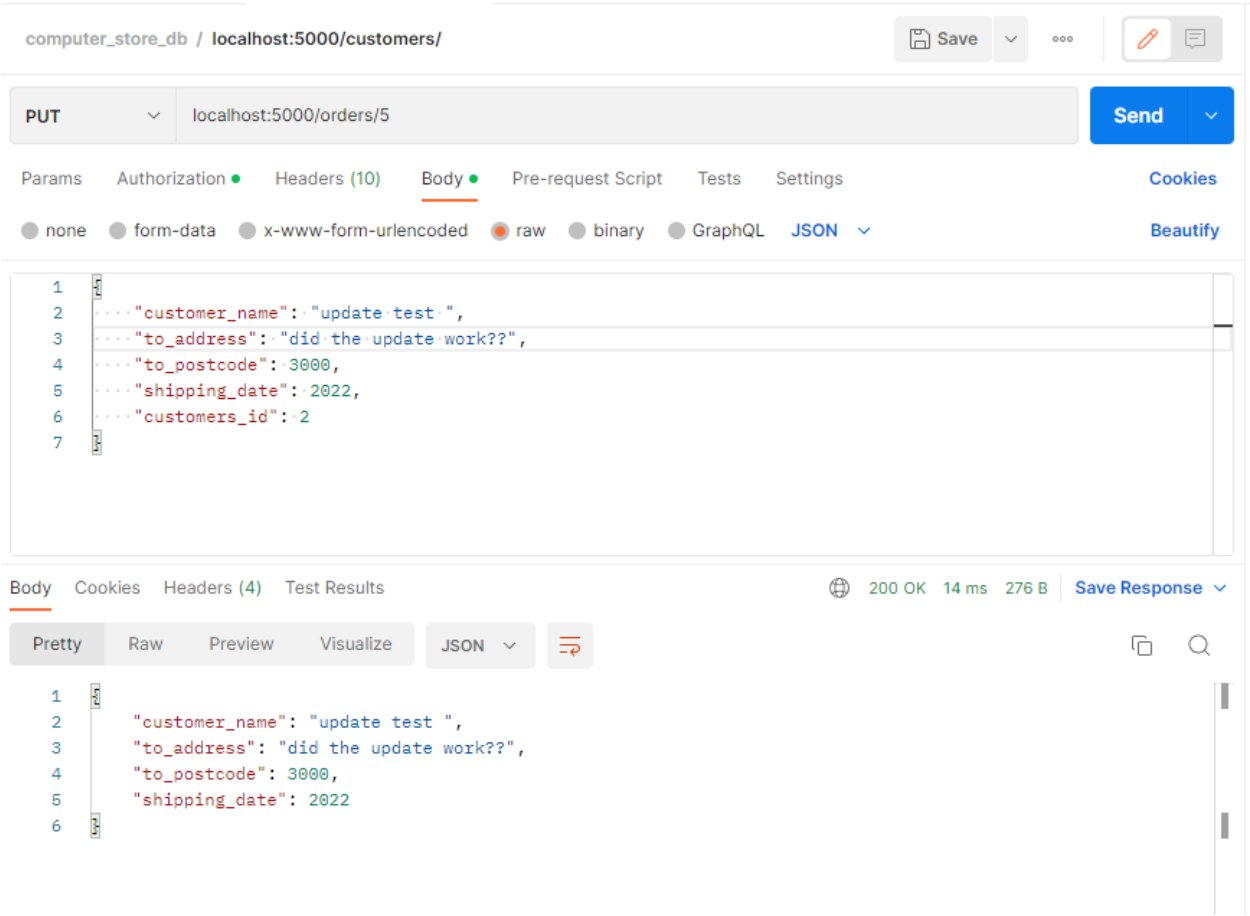
shipping_date: Integer

customers_id: Integer

Expected return

Successful retrieval should return a list of the order associated with the ID you entered at the end of the URL. It will not return the ID.

This method works without the customer ID, unlike the previous method.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/orders/5

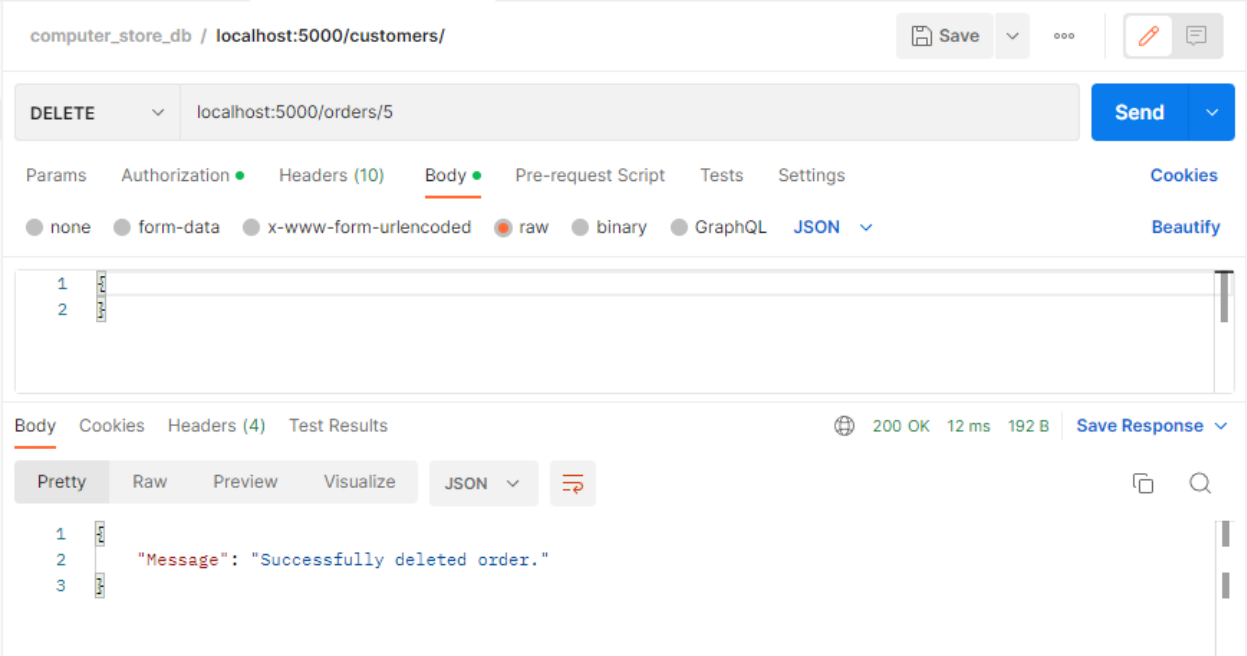
NOTE: The 5 at the end of that link represents the ID of the 5th order. (The one we created, updated and will now delete)

Functionality and Required Fields:

This method takes no fields. It does require a JWT token from an administrator.

Expected return

Successful deletion of the Motherboard should return a message.



Products

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

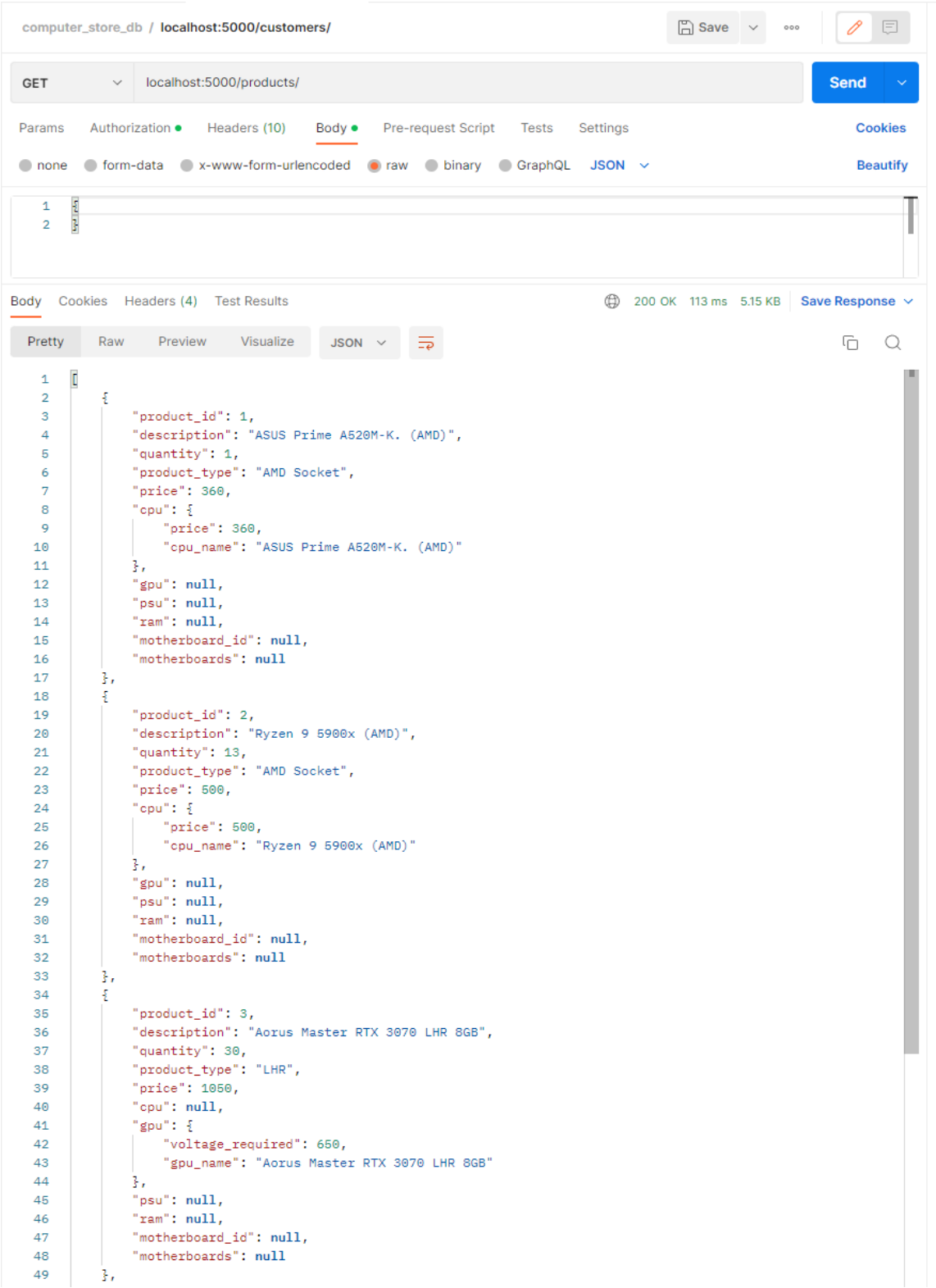
Request ADDRESS = localhost:5000/products/

Functionality and Required Fields:

This method takes no fields. It is compatible with a simple query string as well. See next page for more information.

Expected return

Successful retrieval should return a list of all the products.




```
[
    {
        "product_id": 1,
        "description": "ASUS Prime A520M-K. (AMD)",
        "quantity": 1,
        "product_type": "AMD Socket",
        "price": 360,
        "cpu": {
            "price": 360,
            "cpu_name": "ASUS Prime A520M-K. (AMD)"
        },
        "gpu": null,
        "psu": null,
        "ram": null,
        "motherboard_id": null,
        "motherboards": null
    },
    {
        "product_id": 2,
        "description": "Ryzen 9 5900x (AMD)",
        "quantity": 13,
        "product_type": "AMD Socket",
        "price": 500,
        "cpu": {
            "price": 500,
            "cpu_name": "Ryzen 9 5900x (AMD)"
        },
        "gpu": null,
        "psu": null,
        "ram": null,
        "motherboard_id": null,
        "motherboards": null
    },
    {
        "product_id": 3,
        "description": "Aorus Master RTX 3070 LHR 8GB",
        "quantity": 30,
        "product_type": "LHR",
        "price": 1050,
        "cpu": null,
        "gpu": {
            "voltage_required": 650,
            "gpu_name": "Aorus Master RTX 3070 LHR 8GB"
        },
        "psu": null,
        "ram": null,
        "motherboard_id": null,
        "motherboards": null
    },
```

Query String Associated with the Products end point.

It is compatible with 1 type of query string, though admittedly, its not coded very well. In order to query string the products list, you can try this in the URL.

http://127.0.0.1:5000/products/?description=ASUS Prime A520M-K. (AMD)

The description = part of the query string is flexible enough to let you enter any description, however, you need to enter the entire description for it to filter correctly. If you do not enter the enter description, it will return and empty list. If you only description =, it will return an unbound local error. Unfortunately, I ran out of time to refine it so it is not user friendly and almost defeats the purpose of a query string, but it does work in specific use cases.

127.0.0.1:5000/products/?description=ASUS%20Prime%20A520M-K.%20(AMD)


```
[
  {
    "product_id": 1,
    "description": "ASUS Prime A520M-K. (AMD)",
    "quantity": 1,
    "product_type": "AMD Socket",
    "price": 360,
    "cpu": {
      "price": 360,
      "cpu_name": "ASUS Prime A520M-K. (AMD)"
    },
    "gpu": null,
    "psu": null,
    "ram": null,
    "motherboard_id": null,
    "motherboards": null
  }
]
```

Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/products/3

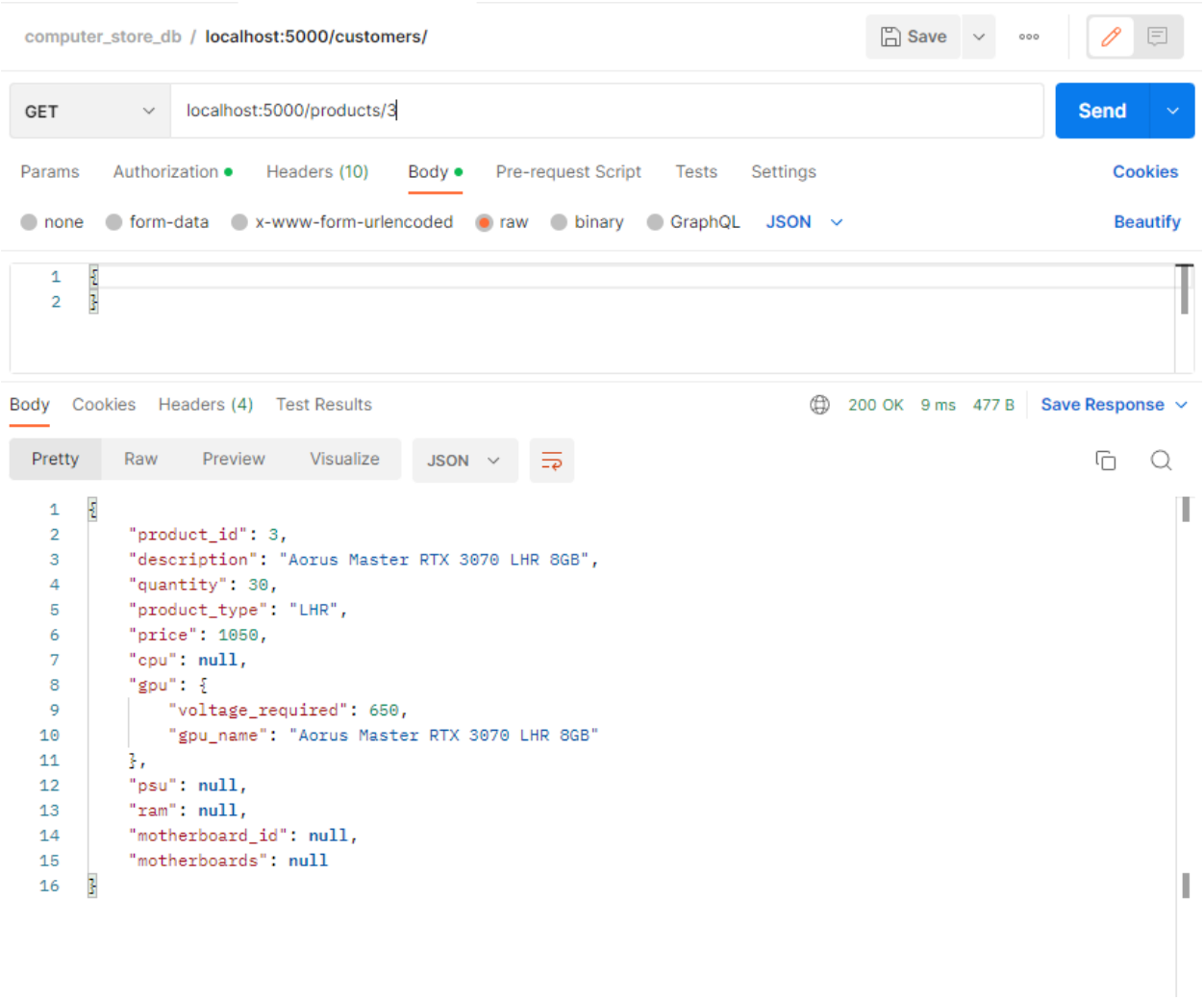
NOTE: The 3 at the end of that link represents the ID of the 3rd product.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the product you are trying to retrieve.

Expected return

Successful retrieval should return a list of all the products.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/products/

Functionality and Required Fields:

Only available to administrators.

description: String

quantity: Integer

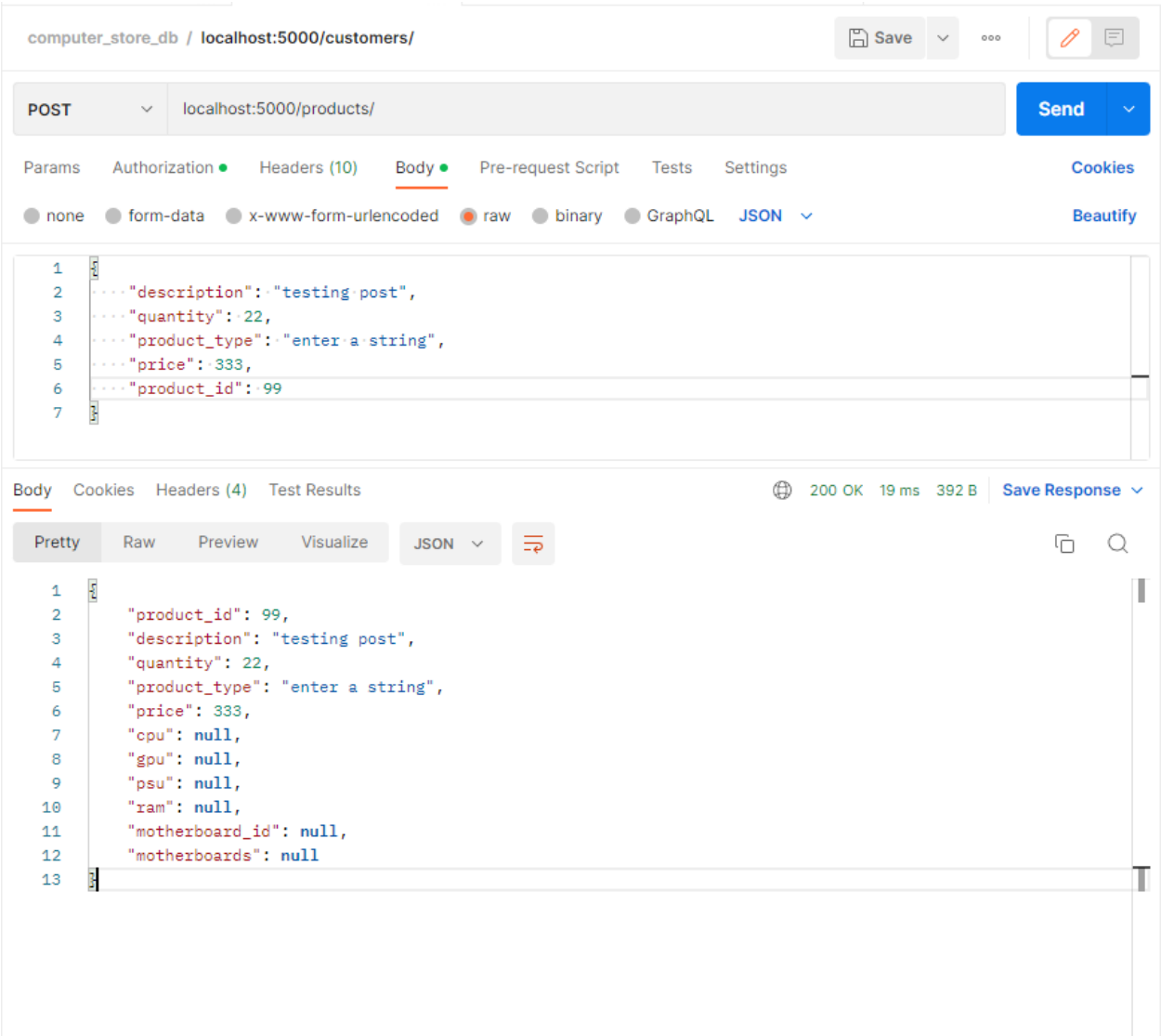
product_type: String

price: Integer

product_id: Integer

Expected return

Successful creation should return the information of the order you entered.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/products/99

NOTE: The 99 at the end of that link represents the ID of the 99th Product. (The one we created and are updating.)

Functionality and Required Fields:

Only available to administrators.

description: String

quantity: Integer

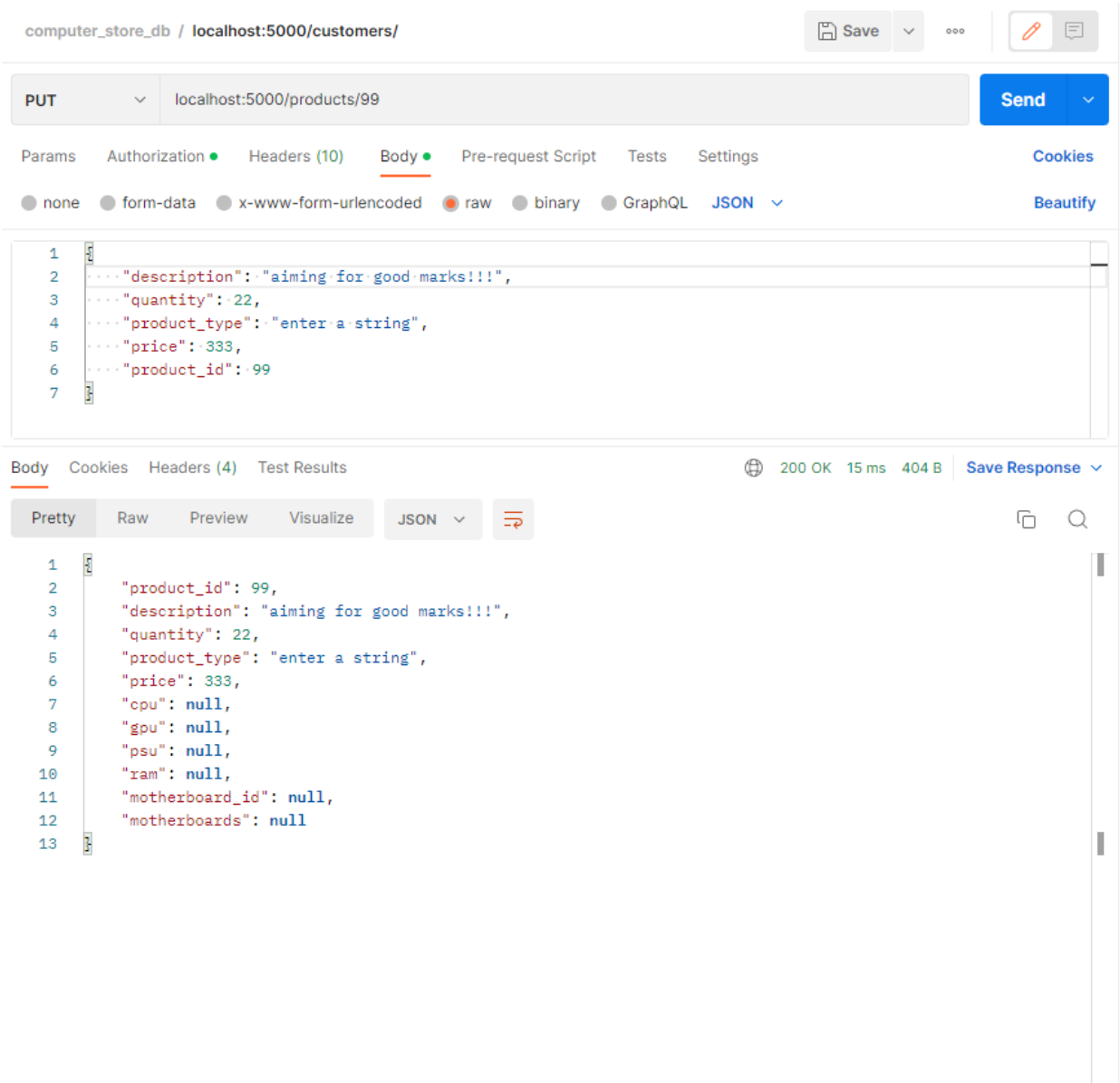
product_type: String

price: Integer

product_id: Integer

Expected return

Successful creation should return the updated information of the order you entered.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/products/99

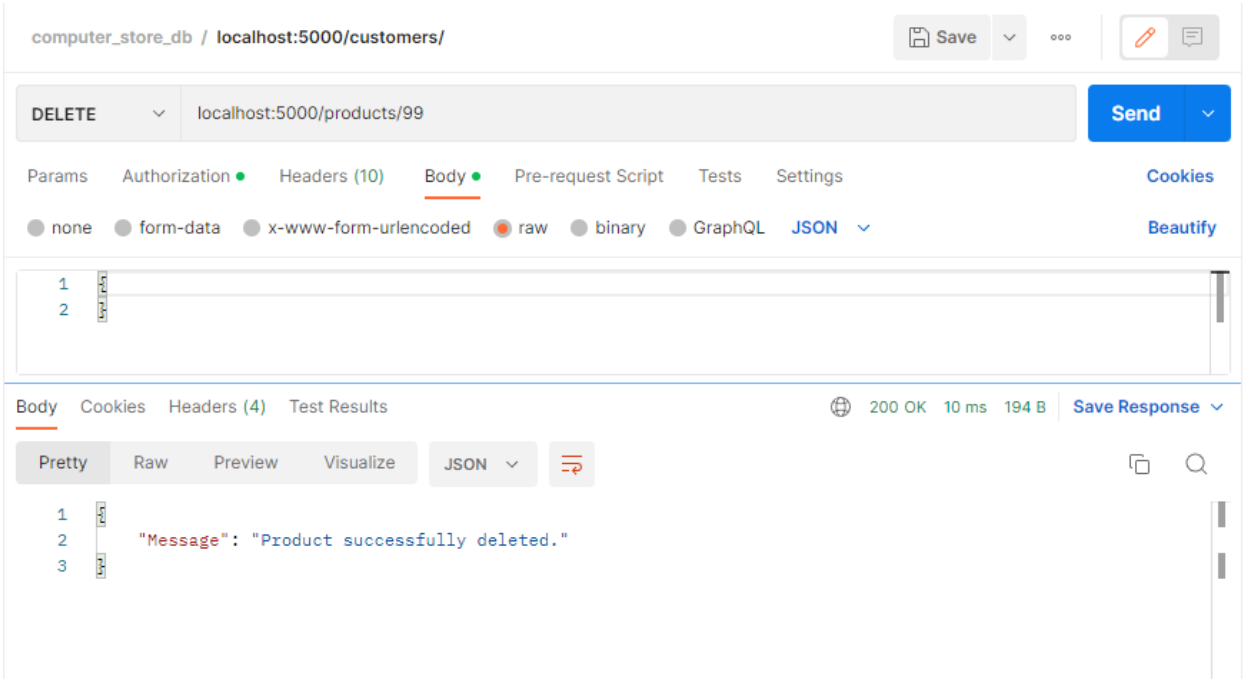
NOTE: The 99 at the end of that link represents the ID of the 99th product.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the product you are trying to retrieve. Administrator access only..

Expected return

Successful deletion of the Product should return a message.



PSU

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

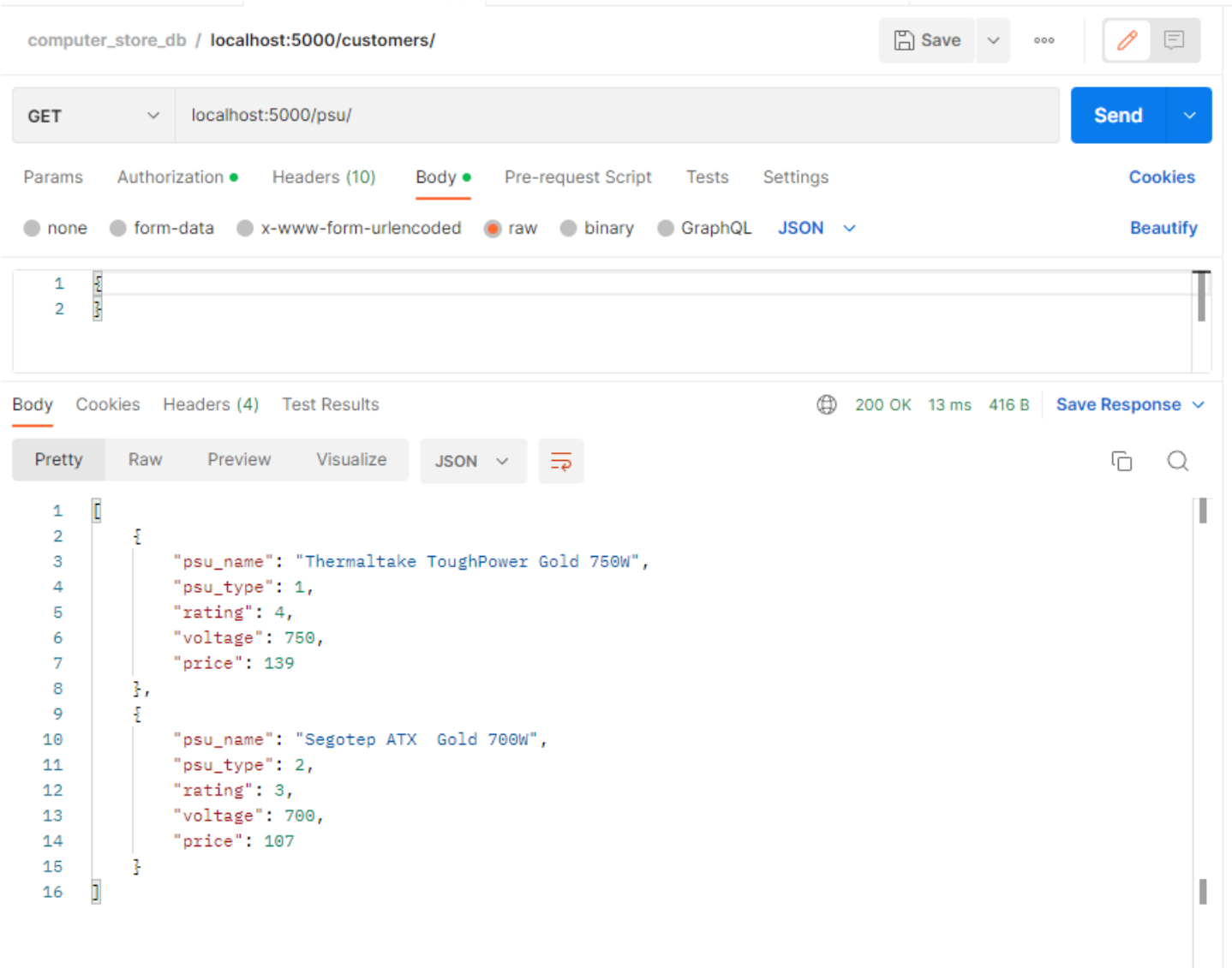
Request ADDRESS = localhost:5000/psu/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the PSUs.



Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/psu/2

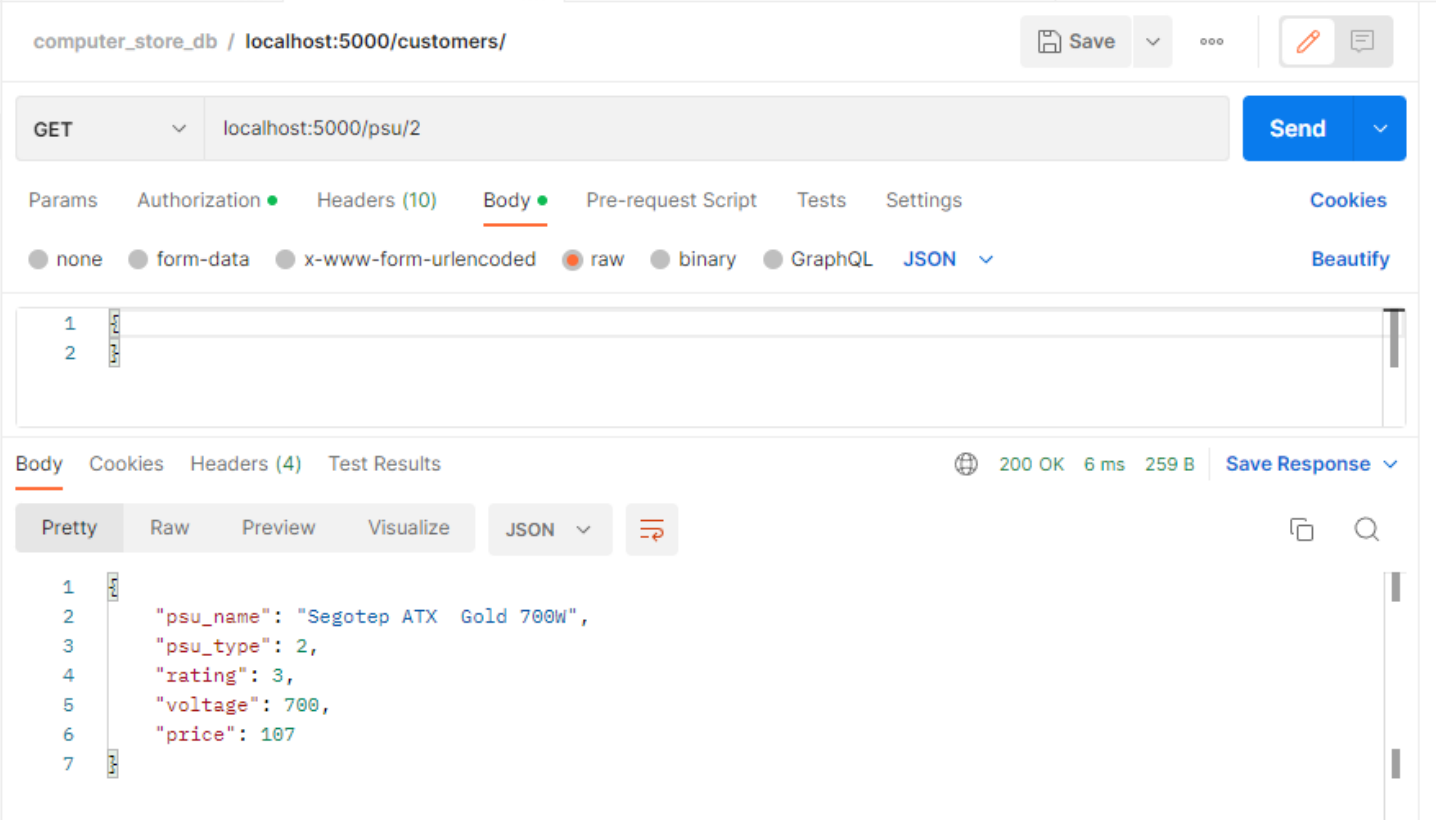
NOTE: The 2 at the end of that link represents the ID of the 2nd PSU.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the PSU you're trying to retrieve.

Expected return

Successful retrieval should return a list of the PSU associated with the ID you entered at the end of the URL.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/psu/

Functionality and Required Fields:

psu_name: String

psu_type: Integer

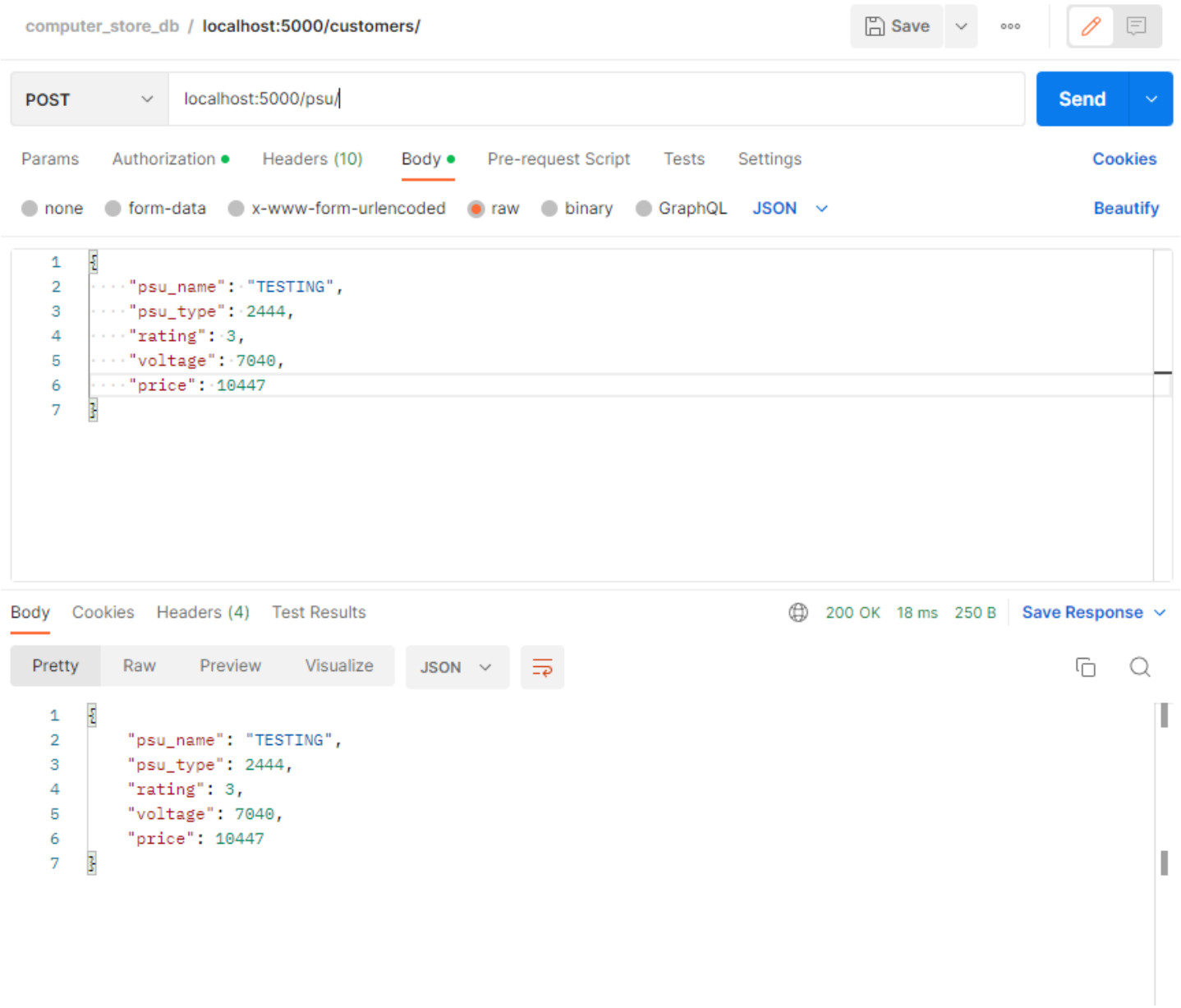
rating: Integer

voltage: Integer

price: Integer

Expected return

Successful retrieval should return a list of the information you entered. It does not return an ID.



Endpoint 4:

Request VERB = PUT

Request ADDRESS = localhost:5000/psu/3

NOTE: The 3 at the end of that link represents the ID of the 3rd PSU.

Functionality and Required Fields:

psu_name: String

psu_type: Integer

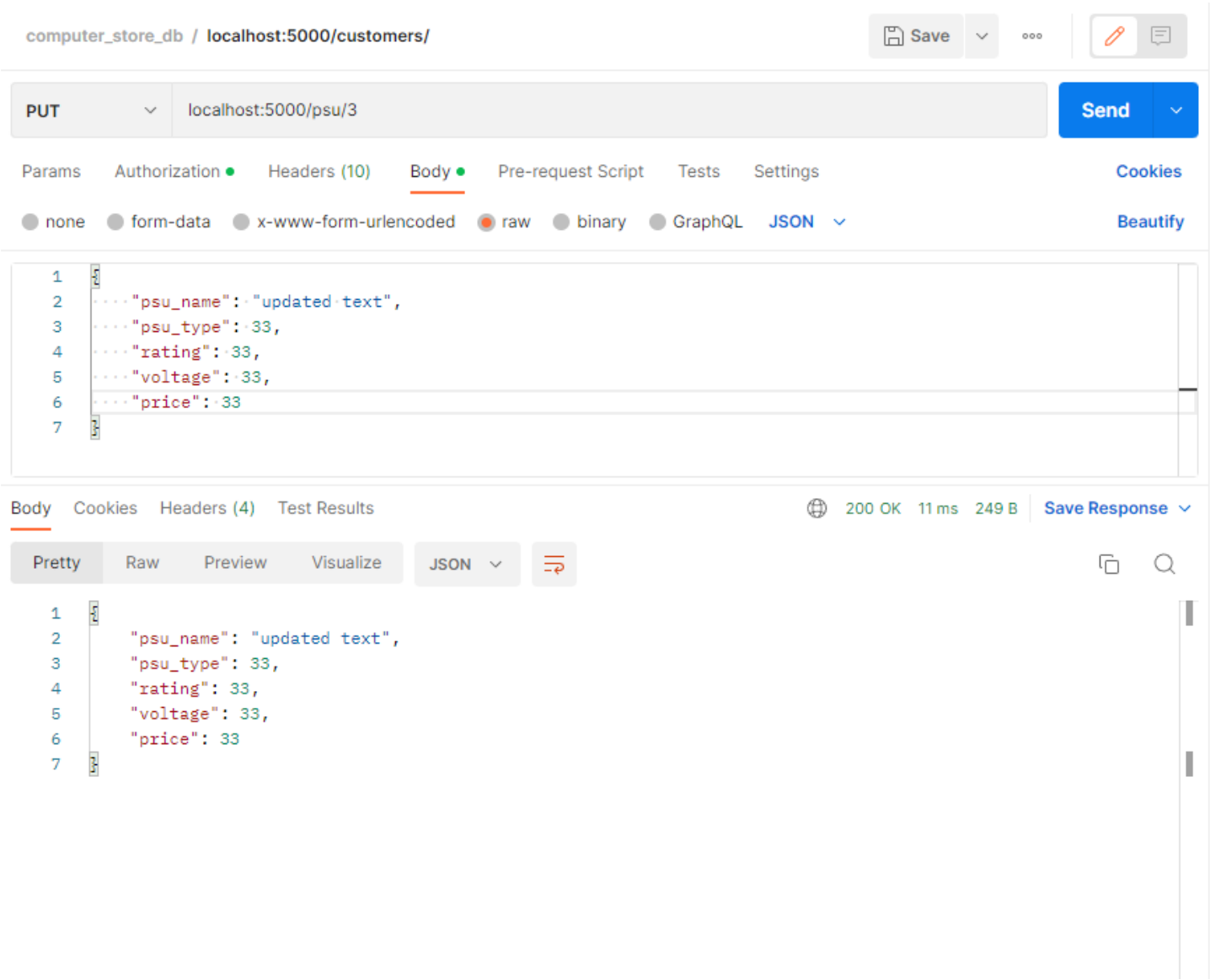
rating: Integer

voltage: Integer

price: Integer

Expected return

Successful retrieval should return a list of the updated information you entered. It does not return an ID.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/psu/3

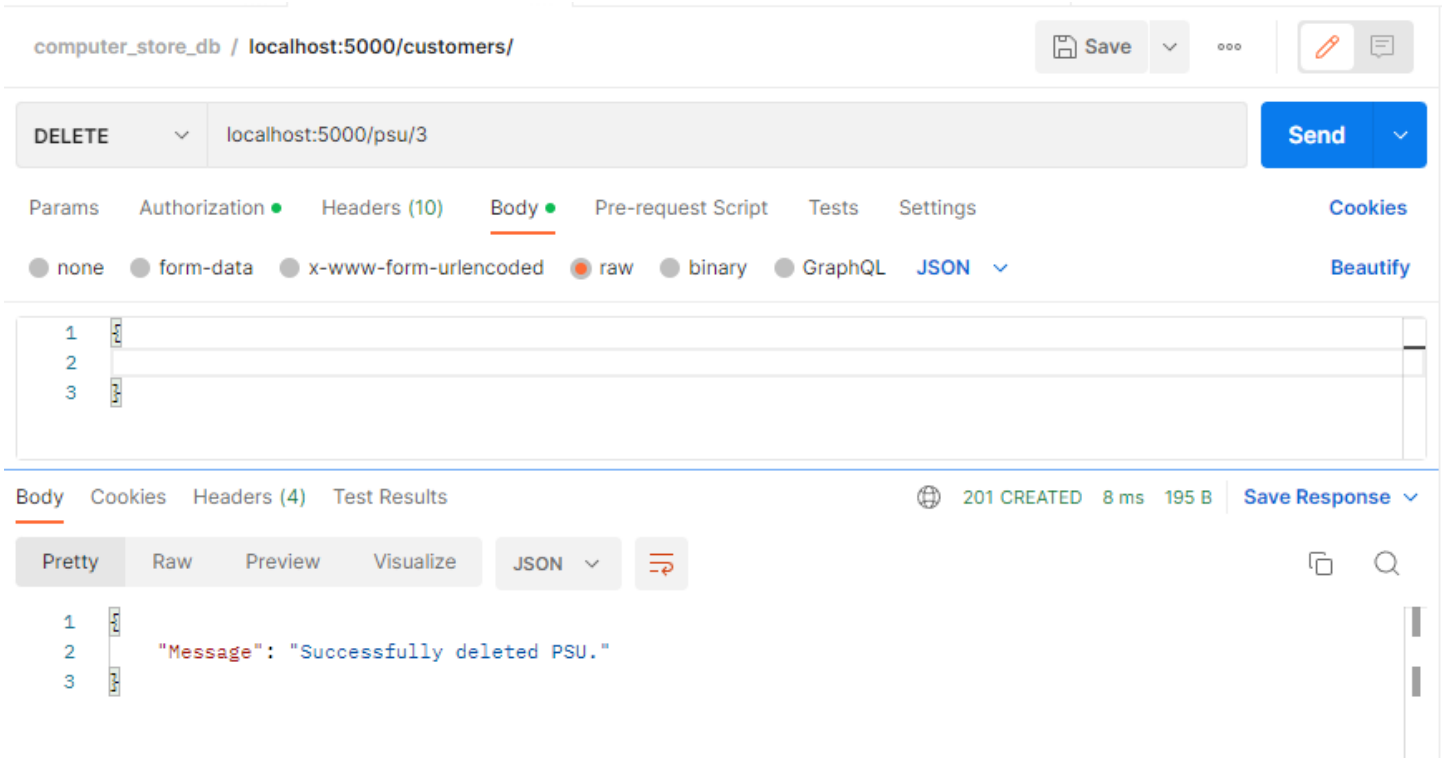
NOTE: The 3 at the end of that link represents the ID of the 3rd PSU.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the PSU you're trying to delete. Admin access only.

Expected return

Successful deletion of PSU should return a message.



RAM

This controller has 5 end points.

Endpoint 1:

Request VERB = GET

Request ADDRESS = localhost:5000/ram/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the RAM.

computer_store_db / localhost:5000/customers/

Save

GET localhost:5000/ram/ Send

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{

}
```

Body Cookies Headers (4) Test Results 200 OK 6 ms 643 B Save Response

Pretty Raw Preview Visualize JSON


```
[
    {
        "ram_type": "DDR4",
        "ram_name": "Team T Force Delta 16GB 3200MHz CL16",
        "rating": 4,
        "ram_id": 1,
        "ram_size": 16,
        "price": 99
    },
    {
        "ram_type": "DDR4",
        "ram_name": "Corsair Dominator 16GB 3200MHz CL16",
        "rating": 5,
        "ram_id": 2,
        "ram_size": 16,
        "price": 165
    },
    {
        "ram_type": "DDR5",
        "ram_name": "Corsair Vengeance 32GB 5600MHz CL36",
        "rating": 5,
        "ram_id": 3,
        "ram_size": 32,
        "price": 319
    }
]
```

Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/ram/2

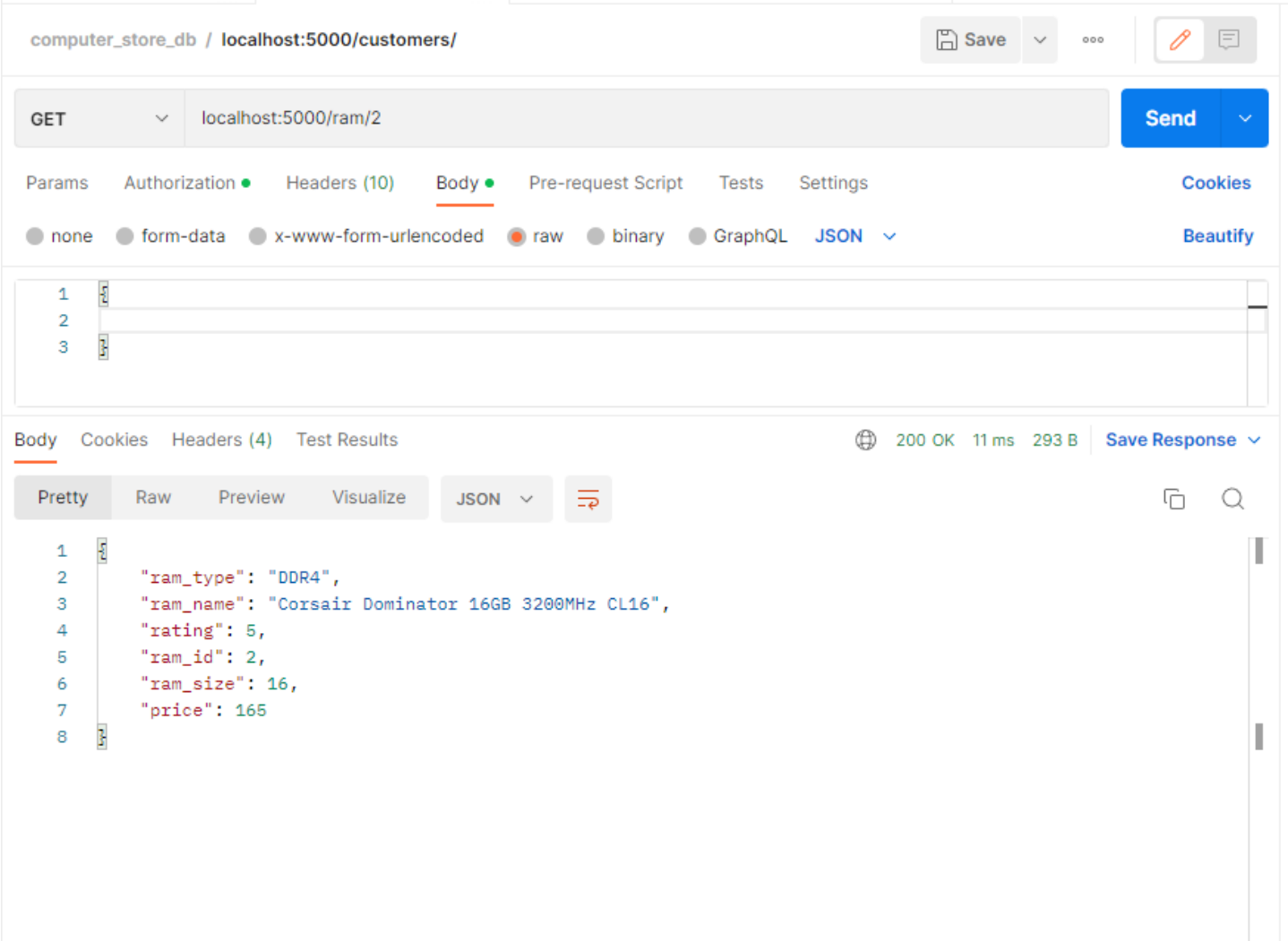
NOTE: The 2 at the end of that link represents the ID of the 2nd Ram.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the RAM you're trying to retrieve.

Expected return

Successful retrieval should return a list of all the RAM associated with the ID you entered in the URL.



Endpoint 3:

Request VERB = POST

Request ADDRESS = localhost:5000/ram/

Functionality and Required Fields:

ram_type: String

ram_name: String

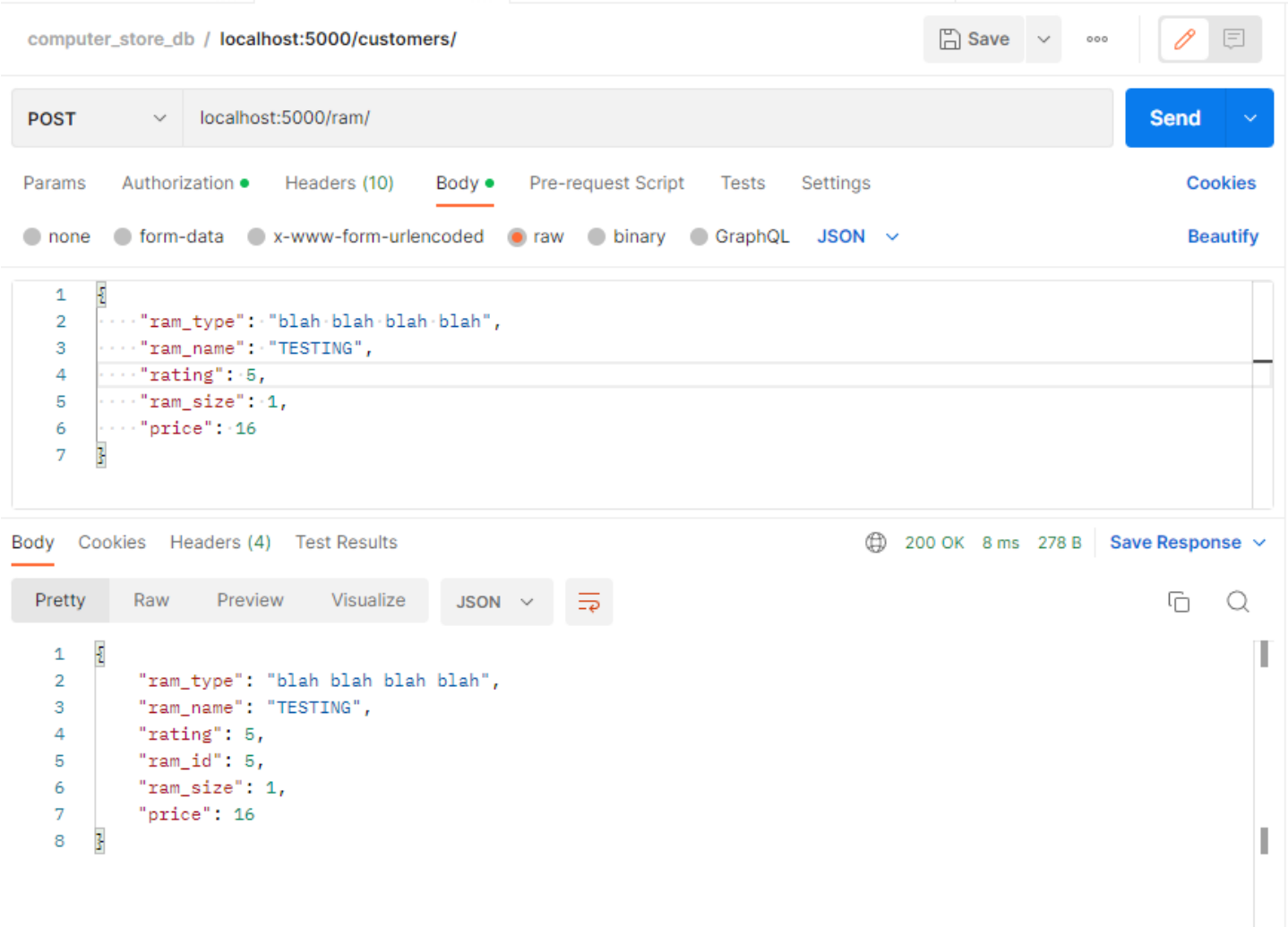
rating: Integer

ram_size: Integer

price: Integer

Expected return

Successful retrieval should return a list of all the information alongside an ID associated to it.



Endpoint 3:

Request VERB = PUT

Request ADDRESS = localhost:5000/ram/5

NOTE: The 5 at the end of that link represents the ID of the 5th Ram.

Functionality and Required Fields:

ram_type: String

ram_name: String

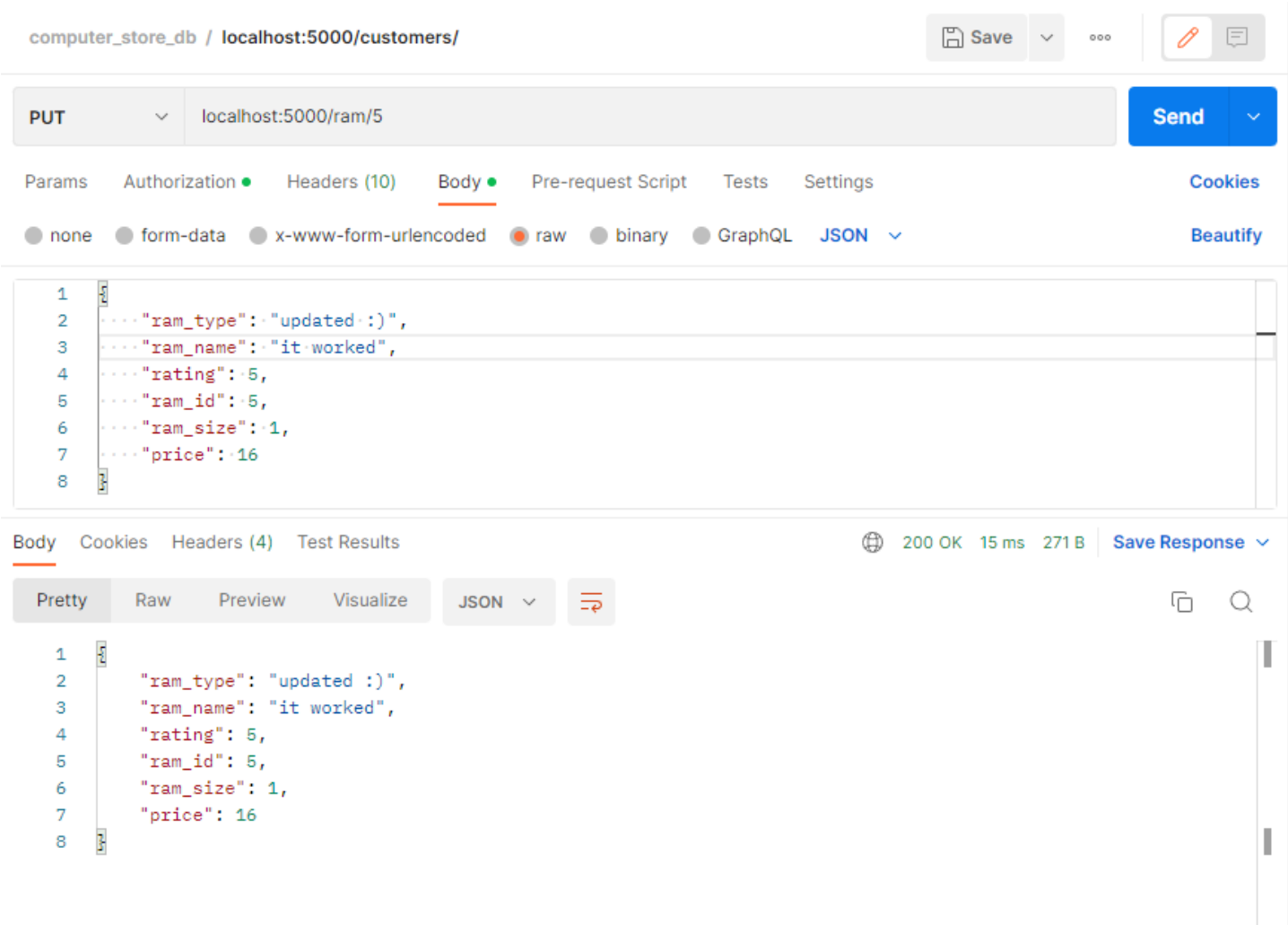
rating: Integer

ram_size: Integer

price: Integer

Expected return

Successful retrieval should return a list of all the updated information alongside an ID associated to it.



Endpoint 5:

Request VERB = DELETE

Request ADDRESS = localhost:5000/ram/5

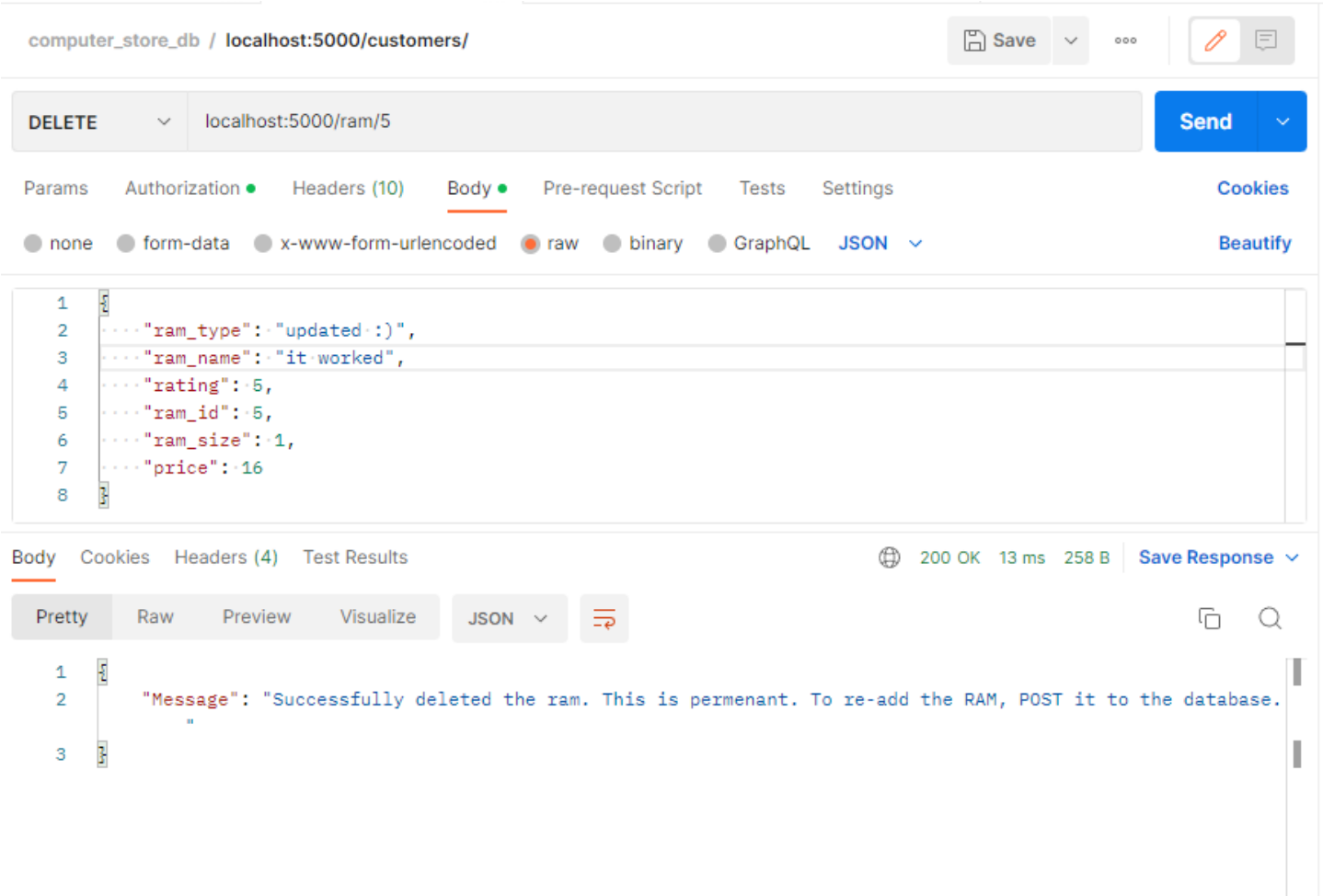
NOTE: The 5 at the end of that link represents the ID of the 5th Ram.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the RAM you're trying to delete.

Expected return

Successful retrieval should return a message.



RATINGS

This controller has 2 end points.

Endpoint 1:

Request VERB = GET

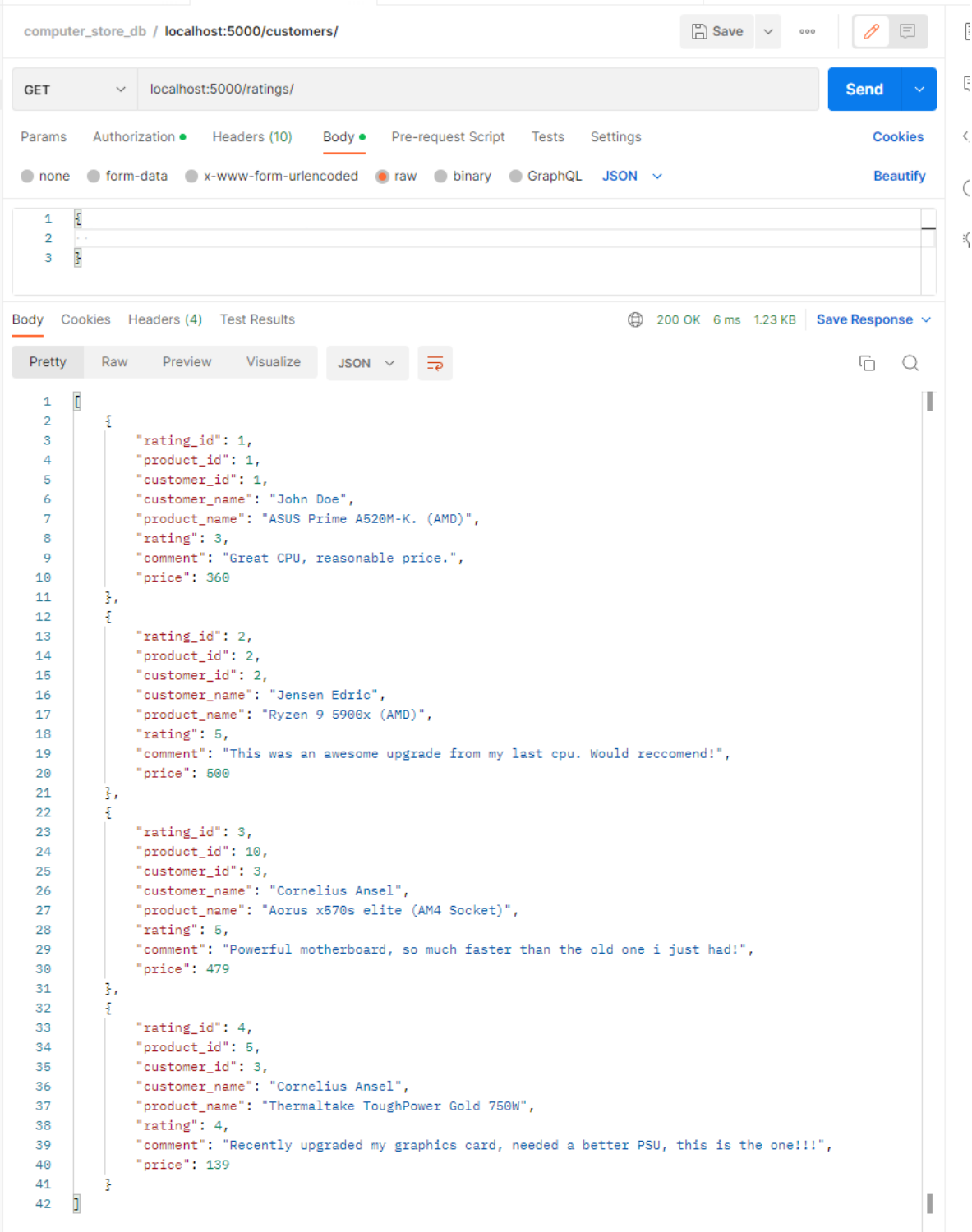
Request ADDRESS = localhost:5000/ratings/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the ratings.




```
[
    {
        "rating_id": 1,
        "product_id": 1,
        "customer_id": 1,
        "customer_name": "John Doe",
        "product_name": "ASUS Prime A520M-K. (AMD)",
        "rating": 3,
        "comment": "Great CPU, reasonable price.",
        "price": 360
    },
    {
        "rating_id": 2,
        "product_id": 2,
        "customer_id": 2,
        "customer_name": "Jensen Edric",
        "product_name": "Ryzen 9 5900x (AMD)",
        "rating": 5,
        "comment": "This was an awesome upgrade from my last cpu. Would reccomend!",
        "price": 500
    },
    {
        "rating_id": 3,
        "product_id": 10,
        "customer_id": 3,
        "customer_name": "Cornelius Ansel",
        "product_name": "Aorus x570s elite (AM4 Socket)",
        "rating": 5,
        "comment": "Powerful motherboard, so much faster than the old one i just had!",
        "price": 479
    },
    {
        "rating_id": 4,
        "product_id": 5,
        "customer_id": 3,
        "customer_name": "Cornelius Ansel",
        "product_name": "Thermaltake ToughPower Gold 750W",
        "rating": 4,
        "comment": "Recently upgraded my graphics card, needed a better PSU, this is the one!!!",
        "price": 139
    }
]
```

Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/ratings/4

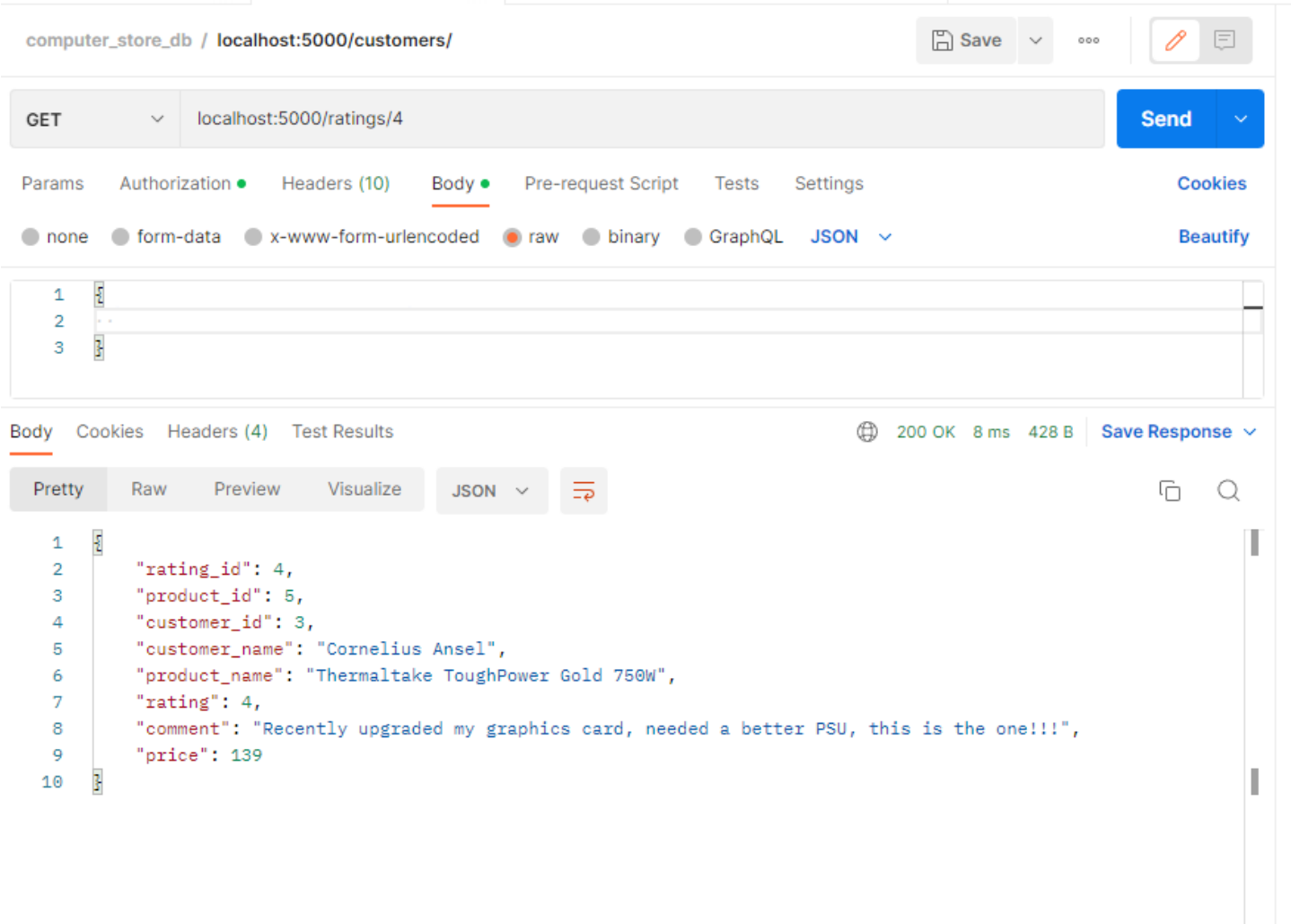
NOTE: The 4 at the end of that link represents the ID of the 4th Rating.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the Rating you're trying to retrieve.

Expected return

Successful retrieval should return a list of all the ratings.



VOLTAGE REQ CONTROLLER

This controller has 2 end points.

Endpoint 1:

Request VERB = GET

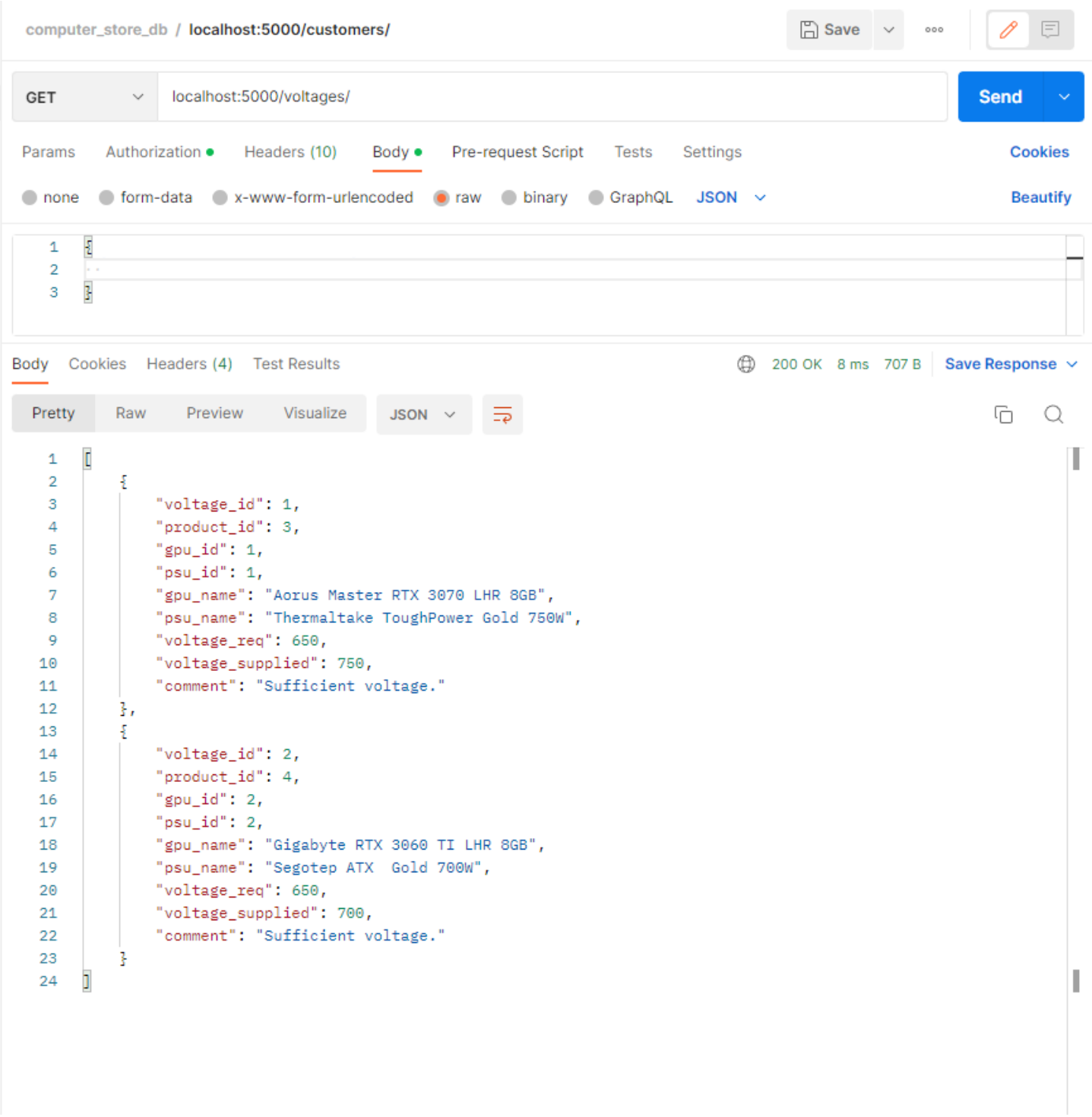
Request ADDRESS = localhost:5000/voltages/

Functionality and Required Fields:

This method takes no fields.

Expected return

Successful retrieval should return a list of all the voltage required instances.



Endpoint 2:

Request VERB = GET

Request ADDRESS = localhost:5000/ratings/4

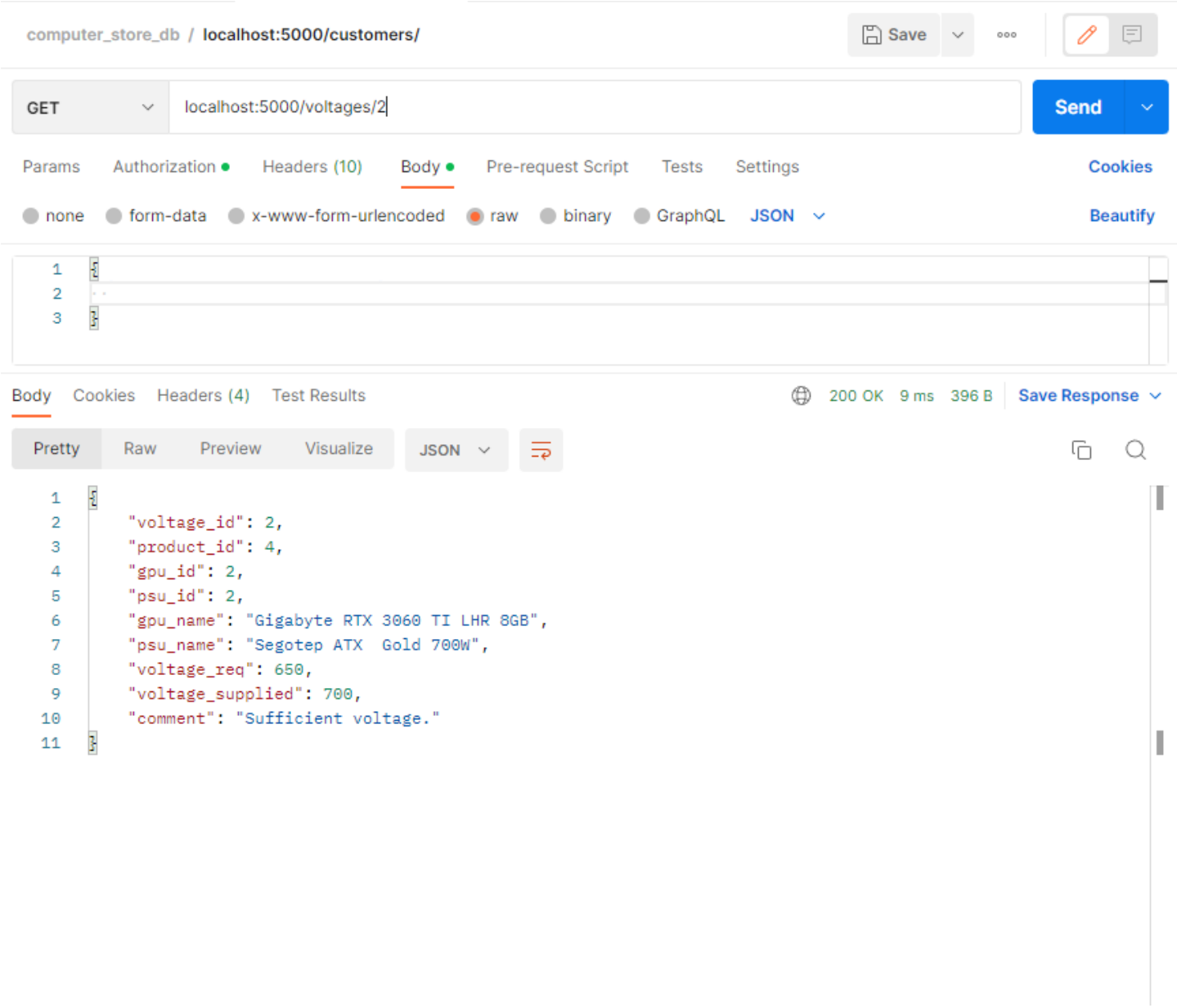
NOTE: The 2 at the end of that link represents the ID of the 2nd Voltage comparison.

Functionality and Required Fields:

This method takes no fields. It only requires the integer ID of the Rating you're trying to retrieve.

Expected return

Successful retrieval should return a list of all the voltage required instances.



Requirement 6: An ERD for your app

The original ERD for my API intended to have the relationships built in a manner that later turned out to be problematic, difficult and also not the best way. Since the design of that ERD, some relationships have changed in term of which tables reference each other and how they reference each other.

A good example is the 'product_id' Foreign Key displayed in the ERD. I had immense difficulties with this approach, the main one being; rather than creating an item twice, I wanted to create it once and have it appear in other tables while simultaneously having a unique ID for both its product instance and its own tables respective instance. To simplify, say I was creating a CPU. I was trying to create an instance of the CPU in the products table so that it could have a Product ID and a CPU ID at the same time.

As I found out very quickly, this approach was awful. I reworked the direction in which relations reference each other. Now, I create a CPU instance, it has a CPU ID. From there, I reference it using CPU ID as a Foreign Key in other tables.

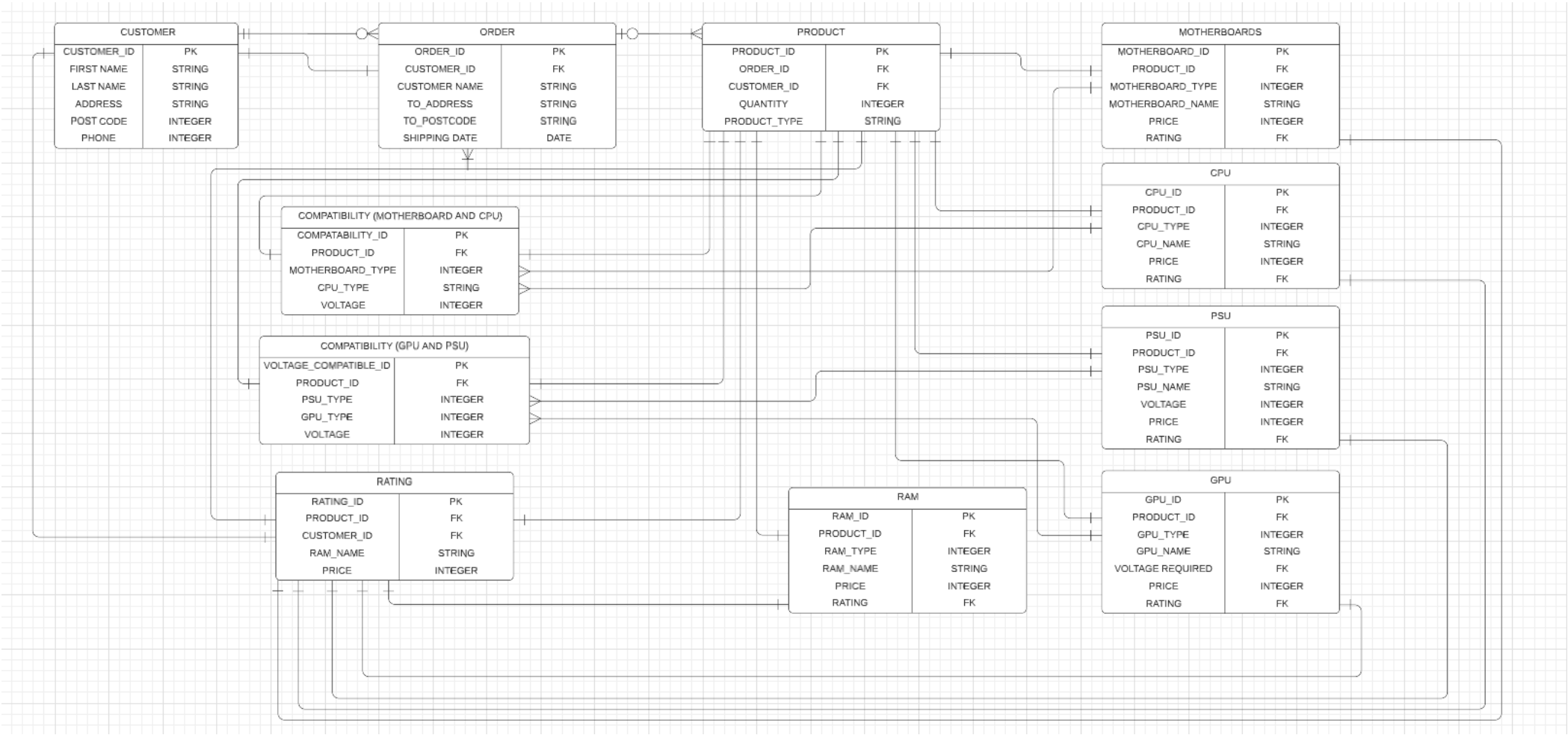
Since the ERD is now redundant, I will provide 2 photos to demonstrate how the tables and keys relate to each other.

models > cpu.py > Cpu

```python
from main import db
#rating will be its own model later. will need to change to foreign key



class Cpu(db.Model):
    __tablename__ = 'cpu'
    cpu_id = db.Column(db.Integer, primary_key=True)
    cpu_type = db.Column(db.Integer)
    cpu_name = db.Column(db.String())
    price = db.Column(db.Integer)
    rating = db.Column(db.Integer)
    product = db.relationship(
        "Product",#class im referencing
        backref ="cpu"# this name can be any, the purpose of this is to use this as a field in the product schema, so make sure it matches with that field, cpu in both would make sense
    )
    compatible = db.relationship(
        "Compat",
        backref ="cpu"
    )
```

models > product.py > Product

```python
from main import db
#rating will be its own model later. will need to change to foreign key



class Product(db.Model):
    __tablename__ = 'product'
    product_id = db.Column(db.Integer, primary_key=True)
    description = db.Column(db.String())
    quantity = db.Column(db.Integer)
    product_type = db.Column(db.String())
    price = db.Column(db.Integer)
    cpu_id = db.Column(db.Integer, db.ForeignKey("cpu.cpu_id"))
    gpu_id = db.Column(db.Integer, db.ForeignKey("gpu.gpu_id"))
    psu_id = db.Column(db.Integer, db.ForeignKey("psu.psu_id"))
    ram_id = db.Column(db.Integer, db.ForeignKey("ram.ram_id"))
    motherboard_id = db.Column(db.Integer, db.ForeignKey("motherboards.motherboard_id"))
```


CUSTOMER
CUSTOMER_ID PK
FIRST NAME STRING
LAST NAME STRING
ADDRESS STRING
POST CODE INTEGER
PHONE INTEGER
ORDER
ORDER_ID PK
CUSTOMER_ID FK
CUSTOMER NAME STRING
TO_ADDRESS STRING
TO_POSTCODE STRING
SHIPPING DATE DATE
PRODUCT
PRODUCT_ID PK
ORDER_ID FK
CUSTOMER_ID FK
QUANTITY INTEGER
PRODUCT_TYPE STRING
MOTHERBOARDS
MOTHERBOARD_ID PK
PRODUCT_ID FK
MOTHERBOARD_TYPE INTEGER
MOTHERBOARD_NAME STRING
PRICE INTEGER
RATING FK
CPU
CPU_ID PK
PRODUCT_ID FK
CPU_TYPE INTEGER
CPU_NAME STRING
PRICE INTEGER
RATING FK
COMPATIBILITY (MOTHERBOARD AND CPU)
COMPATABILITY_ID PK
PRODUCT_ID FK
MOTHERBOARD_TYPE INTEGER
CPU_TYPE STRING
VOLTAGE INTEGER
PSU
PSU_ID PK
PRODUCT_ID FK
PSU_TYPE INTEGER
PSU_NAME STRING
VOLTAGE INTEGER
PRICE INTEGER
RATING FK
COMPATIBILITY (GPU AND PSU)
VOLTAGE_COMPATIBLE_ID PK
PRODUCT_ID FK
PSU_TYPE INTEGER
GPU_TYPE INTEGER
VOLTAGE INTEGER
RATING
RATING_ID PK
PRODUCT_ID FK
CUSTOMER_ID FK
RAM_NAME STRING
PRICE INTEGER
RAM
RAM_ID PK
PRODUCT_ID FK
RAM_TYPE INTEGER
RAM_NAME STRING
PRICE INTEGER
RATING FK
GPU
GPU_ID PK
PRODUCT_ID FK
GPU_TYPE INTEGER
GPU_NAME STRING
VOLTAGE REQUIRED FK
PRICE INTEGER
RATING FK

Requirement 7: Detail any third-party services that your app will use

All imported services listed below:

- Flask
- SQLAlchemy
- Bcrypt
- JWTManger
- JWT Extended (get jwt identity, jwt required, create access token)
- OS
- Blueprint
- Marshmallow
- Validate Length (Marshmallow)
- Fields (Marshmallow)
- Jsonify
- Marshmallow Exceptions (Validation Error)
- Request
- Date Time (Time Delta)

Flask

Used as a framework, responsible for initialising my API and connecting/creating routing for my controllers etc

SQLAlchemy

Used to facilitate communication between my python programs and my PostgreSQL database. Also used as my Object Relational Mapper.

Bcrypt

Used for its password hashing and UTF-8 decoding capabilities. (Used for my Administrator models and instances)

JWTManger

This library was used to store and retrieve tokens.

JWT Extended (get jwt identity, jwt required, create access token)

Used to create the tokens that will be provided to Administrators. Also used to add restrictions to my controllers and specific HTTP verbs so that I can control who can access certain routes.

OS

All libraries and modules were installed in a virtual environment, to access Operating System capabilities, I imported OS.

Blueprint

Blueprint was used to create reusable code and further define routes used by my controllers.

Marshmallow

Used to convert simple and complex data types from (and to) python and JSON.

Validate Length (Marshmallow)

Used to enforce minimum length criteria for passwords as defined in the schemas.

Fields (Marshmallow)

Used in the context of method fields, whereby a function is defined to store an object, the object is returned to be serialized into a JSON format.

Jsonify

Used to serialise data into JSON format.

Marshmallow Exceptions (Validation Error)

Used to catch and return Error Code messages for a range of reasons for my controllers. For example, if a user enters an Integer ID at the end of the URL when performing a GET request for customers.

If the database does not have that information, it can now return a 404 not found. Or if a user does not have administrator privileges on the DB, it can now return a 403 forbidden.

Request

Used to read the request body and return it in a resolved manner after parsing the body of the request as JSON.

Date Time (Time Delta)

Used to gain access to classes to manipulate dates and times.

NOTE:

All packages, modules, libraries etc were installed in a virtual environment. A copy of the required items was made in the requirements.txt file. Install all of the dependencies before using the API.

Requirement 8: Describe your projects models in terms of the relationships they have with each other

ADMIN

Primary Key

- admin_id

Relationship: No relationship to other tables, Administrators are their own entities.

This model does not relate to any other models. Its entire use case is for the authentication and access of routes defined in the controllers and protected with tokens.

COMPAT

Primary Key

- compat_id (PK)

Foreign Key Constraints

- cpu_id
- motherboard_id

Relationship: Many to Many. (Many CPUs and Motherboards to many combabilities)

This model is used to check the compatibility of a CPU and a Motherboard. Compatibility in this API is defined by socket type. For example, an Intel CPU will not be compatible with an AM4 motherboard.

CPU

Primary Key

- cpu_id

Referenced by

- compat table
- product table

Relationship: One to Many. (1 CPU can be many products, 1 CPU can be in many compatibility checks)

The CPU model has a few instances which are listed as individual products and also referenced by the compatibility model.

CUSTOMERS

Primary Key

- customers_id

Referenced by

- orders table
- rating table

Relationship: One to Many (1 Customer to many orders, 1 Customer to many ratings)

The customer's model doesn't have a very large impact in the grand scheme of things, I implemented it to test out an ordering and review system.

GPU

Primary Key

- gpu_id

Referenced by

- products table
- voltages table

Relationship: One to Many. (1 GPU can be many products, 1 GPU can be in many voltage requirements checks)

GPU instances are listed as individual products and also referenced in the voltage requirements table alongside the PSU.

MOTHERBOARD

Primary Key

- motherboard_id

Referenced by

- compat table
- product table

Relationship: One to Many. (1 Motherboard can be many products, 1 Motherboard can be in many compatibility checks)

The Motherboard model has a few instances which are listed as individual products and also referenced by the compatibility model.

ORDERS

Primary Key

- order_id

Foreign Keys

- customers table

Relationship: Many to Many. (Many orders can be from Many Customers)

Orders were implemented as part of a purchasing and review test. In future this will be redefined to display the product the customer ordered but at the moment it just shows an order instance. I was originally trying to figure out a way to have some of these details hidden behind authentication but I never got that far.

PRODUCTS

Primary Key

- product_id

Foreign Key Constraints

- cpu_id
- gpu_id
- motherboard_id
- psu_id
- ram_id

Products is the model with the most instances. Every item specific to another model is also listed as a product in a collective list. This was a great way for me to provide a 1 stop shop menu type of list to users.

Referenced by

- ratings table
- voltages table

Relationship: Many to One (Many products can be 1 item, ie many products can be the 1 CPU)

PSU

Primary Key

- psu_id

Referenced by

- products table
- voltages table

Relationship: One to Many (1 PSU can be Many products, 1 PSU can be in many Voltage requirements checks)

PSU instances are listed as individual products and also referenced in the voltage requirements table alongside the GPU.

RAM

Primary Key

- ram_id

Referenced by

- product table

Relationship: One to Many (1 RAM can be Many products)

RATINGS

Primary Key

- rating_id

Referenced by

- customers table
- product table

Relationship: Many to One (Many Ratings can be from One Customer)

VOLTAGES

Primary Key

- voltage_id

Foreign Key Constraints

- gpu_id
- product_id
- psu_id

Relationship: Many to Many (Many checks can be from Many Products)

Requirement 9: Discuss the database relations to be implemented in your application

Relations are defined in my database via Primary and Foreign Keys. The implementation of these keys creates unique data which can be used to relate against other data in the database.

My API has 12 models, each of these models, schemas and controllers and therefore 12 relations.

Each of these relations have their own information which is separate from the rest of the program and each of them also contain information borrowed or sourced from each other.

To elaborate, the voltage requirements model is used to determine if the Voltage draw of a GPU is less than the Voltage output of a PSU. It contains its own field such and a description, an ID of the comparison and whether or not the items are compatible. It also contains information that has been defined about both of those models in other tables.

Implementing a structure like this allows for consistent data integrity.

Pictured below is the model for the Voltage Requirement comparison. As you can see, it references 3 other independent tables for information. These tables are the product table, the GPU table and the PSU table.

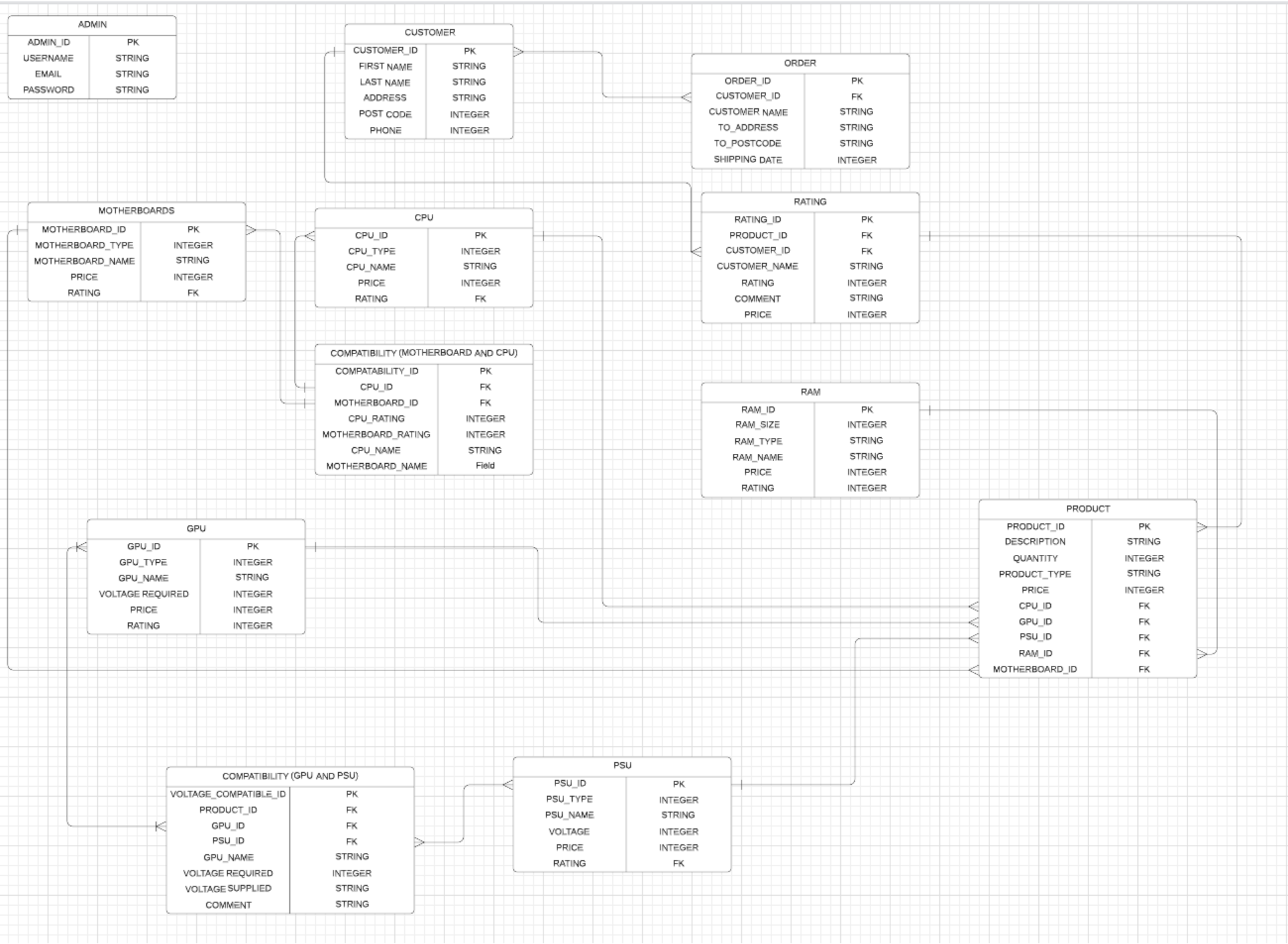
```
schemas > voltage_req_schema.py > ...
from main import ma

class VoltageSchema(ma.Schema):
    class Meta:
        ordered = True
        fields = ['voltage_id', 'product_id', 'gpu_id', 'psu_id', 'gpu_name', 'psu_name', 'voltage_req', 'voltage_supplied', 'comment']

voltage_schema = VoltageSchema()
voltages_schema = VoltageSchema(many = True)
```

```
    psu_id = db.Column(db.Integer, db.ForeignKey('psu.psu_id'))
    gpu_name = db.Column(db.String())
    psu_name = db.Column(db.String())
    voltage_req = db.Column(db.Integer)
    voltage_supplied = db.Column(db.Integer)
    comment = db.Column(db.String())

    #need to check if voltage references work as intended.
    #foreign key function here is supposed to check voltage needed against the voltage we have available.
```

As defined here in the schema, the information it collects from other tables can be displayed in the browser (in response to a GET request) as a means to provide clear, accurate and unique data.

```
schemas > voltage_req_schema.py > ...
from main import ma

class VoltageSchema(ma.Schema):
    class Meta:
        ordered = True
        fields = ['voltage_id', 'product_id', 'gpu_id', 'psu_id', 'gpu_name', 'psu_name', 'voltage_req', 'voltage_supplied', 'comment']

voltage_schema = VoltageSchema()
voltages_schema = VoltageSchema(many = True)
```

These relations are consistent throughout the entire API, to better describe them, here is an updated and accurate ERD.


ADMIN
ADMIN_ID PK
USERNAME STRING
EMAIL STRING
PASSWORD STRING
CUSTOMER
CUSTOMER_ID PK
FIRST NAME STRING
LAST NAME STRING
ADDRESS STRING
POST CODE INTEGER
PHONE INTEGER
ORDER
ORDER_ID PK
CUSTOMER_ID FK
CUSTOMER NAME STRING
TO_ADDRESS STRING
TO_POSTCODE STRING
SHIPPING DATE INTEGER
MOTHERBOARDS
MOTHERBOARD_ID PK
MOTHERBOARD_TYPE INTEGER
MOTHERBOARD_NAME STRING
PRICE INTEGER
RATING FK
CPU
CPU_ID PK
CPU_TYPE INTEGER
CPU_NAME STRING
PRICE INTEGER
RATING FK
RATING
RATING_ID PK
PRODUCT_ID FK
CUSTOMER_ID FK
CUSTOMER_NAME STRING
RATING INTEGER
COMMENT STRING
PRICE INTEGER
COMPATIBILITY (MOTHERBOARD AND CPU)
COMPATABILITY_ID PK
CPU_ID FK
MOTHERBOARD_ID FK
CPU_RATING INTEGER
MOTHERBOARD_RATING INTEGER
CPU_NAME STRING
MOTHERBOARD_NAME Field
RAM
RAM_ID PK
RAM_SIZE INTEGER
RAM_TYPE STRING
RAM_NAME STRING
PRICE INTEGER
RATING INTEGER
PRODUCT
PRODUCT_ID PK
DESCRIPTION STRING
QUANTITY INTEGER
PRODUCT_TYPE STRING
PRICE INTEGER
CPU_ID FK
GPU_ID FK
PSU_ID FK
RAM_ID FK
MOTHERBOARD_ID FK
GPU
GPU_ID PK
GPU_TYPE INTEGER
GPU_NAME STRING
VOLTAGE REQUIRED INTEGER
PRICE INTEGER
RATING INTEGER
COMPATIBILITY (GPU AND PSU)
VOLTAGE_COMPATIBLE_ID PK
PRODUCT_ID FK
GPU_ID FK
PSU_ID FK
GPU_NAME STRING
VOLTAGE REQUIRED INTEGER
VOLTAGE SUPPLIED STRING
COMMENT STRING
PSU
PSU_ID PK
PSU_TYPE INTEGER
PSU_NAME STRING
VOLTAGE INTEGER
PRICE INTEGER
RATING FK

Requirement 10: Describe the way tasks are allocated and tracked in your project

ERD planning and Setbacks

In order to plan and track tasks needed to develop this API, I first created an ERD. Doing this allows me to visually map out potential relationships between my models and allows me to quickly and easily change my approach before ever writing a single line of code.

The ERD went through a few different revisions, both of which have been provided in this document. Despite planning it this way, I still managed to plan the API development in such a manner that I didn't realise potential short coming or challenges until I actually stumbled into them

As seen in the original ERD, my relationships were planned in a way that seemed functional at the time but actually caused a series of problems such as:

- unclear relationships between models
- tables relying on information from areas that I was unable to provide it to
- double handling of instances in order to enter the data into 2 different tables
- non functional schemas (due to foreign keys being defined incorrectly and the information not being accessible)

This cost me a lot of time and I actually had to stop and go back to the drawing board. After some much-needed consultation with Jairo, I was able to understand a far superior approach to mapping out my instances, models, relationships and keys.

When the approach was understood, I was able to drop the database all together and work on restricting my models. After the restructure was made, I moved on to instances. Doing it in this order allowed me to add data to these relations and further reference actual data as I progressed.

Incorporation of Git for Version Control

In accordance with best practices, I sectioned the building progress into multiple branches. This afforded me the luxury of testing out features and establishing new relationships for my database whilst also having a previous, functional commit to return to if things went south again.

There were a total of 5 branches used for the development of this API:

- master
- models
- restructure
- controllers
- methods

They were created and developed under in that order as well. The order of the branches actually tells a small story about the setback I faced during development.

Task Order and Allocation

In order for me to have real data to work with and test, I needed to establish my models first. Note, by the time I was creating models, I had already established my environment variables and linked my database to the application.

After the model structure was defined, I created an instance for that model. This allowed me to see the basic structure of my first relation. From there, in order to make sure I didn't miss anything, I created the corresponding schema. Once I had a few models defined and their instances implemented, I felt more confident testing out Foreign Keys. Linking my tables with Foreign Keys was challenging at first because it felt like a much more complicated concept than I had originally thought. After some practise, the process ended up being very reasonable and not too challenging.

Once I reached a stage where I had a decent chunk of data in PostgreSQL, I decided to put my schemas to work and began designing and building my controllers. With the help of Flask Blueprint, I was able to define a reusable "mould" to which my controllers would conform.

This saved a tremendous amount of time and helped me design clear, consistent controllers. A direct result of this effort meant that debugging controllers was not difficult and incredibly valuable to my education.

As it stands for submission, the API is functional and returns data correctly. I am proud of it. However, I am very conscious of the shortcomings that I did not have time to resolve. I am aware of most of the issues I have faced and will list them on the next page.

Unfixed problems/Shortcomings of my API

My API does a lot of things well, but there is a number of shortcomings I am aware of and intend to fix in my own time after submission. Unfortunately, I just ran out of time or am not quite skilled enough to tackle these problems yet. Here is an honest list of the shortcomings/known issues:

- The only Token that grants administrator access is generated from the hardcoded administrator instance. Using the POST method to create an admin will generate a token, that token is just not recognised by my API and will deny access to the routes.

```
#CREATE ORIGINAL ENTRY FOR ALL TABLES. REMAINDER OF THE DATA WILL BE ADDED VIA POST METHOD.
def seed_db():

    admin1 = Administrator(
        username = 'Chris',
        email = 'chris@admin.com',
        #encrypt password
        password = bcrypt.generate_password_hash('Weeeeeeeee').decode('utf-8')
    )
    db.session.add(admin1)
```

- The Query String I coded for the Products needs to be so specific that it literally defeats the purpose of a Query String. It does work but, with the effort put into writing the full description of the product, you're better off using a GET request with an ID.

- Despite defining data types in my schema, some fields still allow the user to input the wrong data type, I tried to fix this for some time but couldn't quite resolve it completely. This happens in accordance with required fields, so I think I've narrowed problem down to those lines of code. You will be required to provide a value, the accuracy of the value you provide will not always matter

- I accidentally pushed pycache files to GitHub, I then continued to work on the project for some time and pushed more pycache files to GitHub. During this time, I had edited specific lines in both branches and so the pycache files conflict when I try to merge. I haven't resolved this yet and as such, to see the most up to date version of the API, please use it from the "methods" branch.

- The voltage comparisons for the GPU and PSU instances are not calculated mathematically. I hard coded these results into database and as a consequence, I have not implemented a POST method for this entity.

While these shortcomings are not the end of the world, they are issues I would like to fix. Over time I intend to continue working on this API to resolve these issues.

Attribution

Most questions have been answered by me using either class knowledge etc or anecdotal evidence/opinions.

Question 3:

https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-UNIQUE-CONSTRAINTS

https://www.postgresql.org/docs/current/sspi-auth.html

https://pgdash.io/blog/scaling-postgres.html