**Lab 1 - REST, SQL and ORMS**

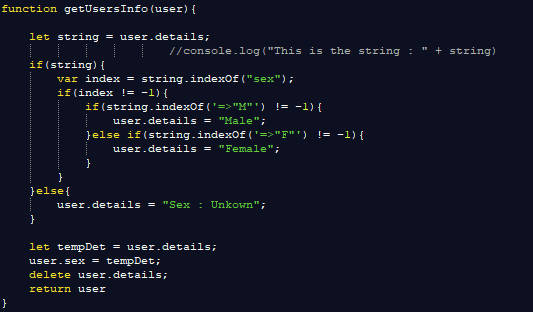
I have set up my environment on a Digital Ocean Ubuntu Linux instance. Here I have hosted NodeJS , all of my source code and also the PSQL database.

I have followed the setup steps and also pulled the example database for this lab as instructed.

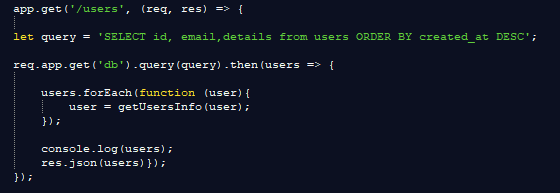
**Problem Sets.**

* 1. **GET /users :**

**getUserDetails() function used for part (a) and (b):**



GET/users request :

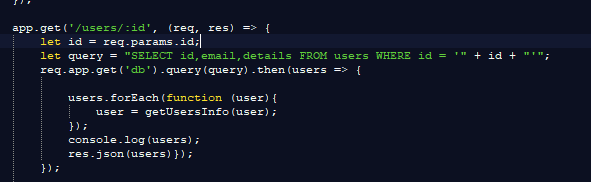


**Result :** On IPaddress/users

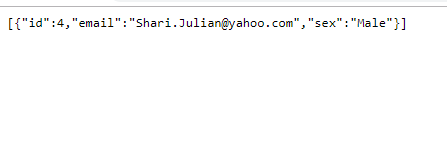


* 1. **Get /user/:id : GET API CODE**

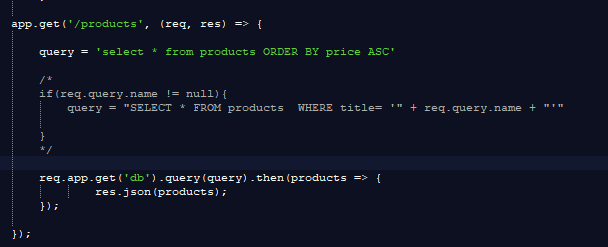
Uses getUserInfo() function from above



**Result :** On IPaddress/users/4 (:id)



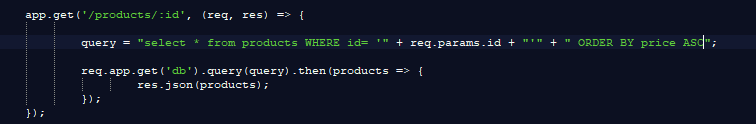
* 1. **Get/products**



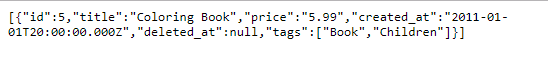
**Result :**



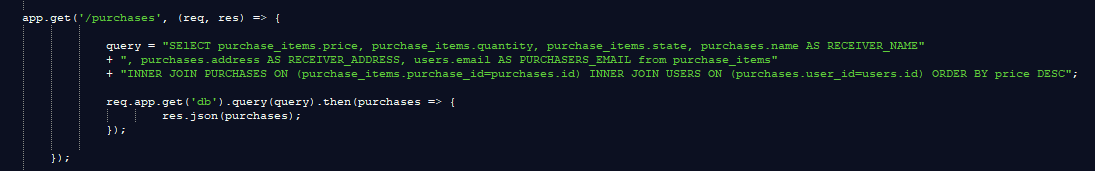
* 1. **Get/products/:id**



Result :



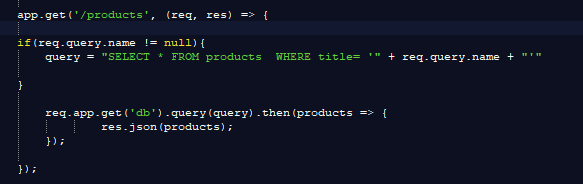
* 1. **Get/purchases**



Result :



1. 1. **Get/products[?name=string]**

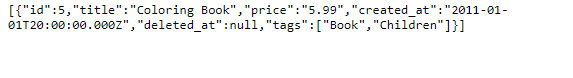


SQL INJECTION METHODS :

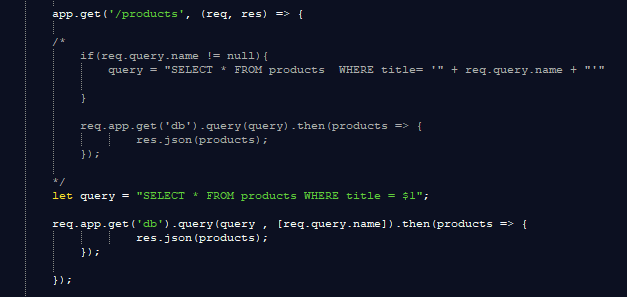
If you put that into the ?name = name ; (INJECTION)\_\_\_\_\_\_\_\_\_\_\_

You can end the current query that you are making and insert your own one as it will append itself to the query string inside the if statement. You can do anything to the database at that point that SQL allows.

For example : ;DELETE FROM products WHERE id=5; or DELETE \* FROM products



1. 1. **Parameterised Query :**



1. Prepare sql statement using let query = “Statement and $1”
   1. $1 is a variable that will be passed in from the parameter inserted by API consumer
   2. Pass this prepared query into the database with the parameter [parameter name] (This works and prevents sql injection because its replaces the $1 variable with your query that is also automatically escaped hence preventing sql injection.)
   3. **Stored procedure :**

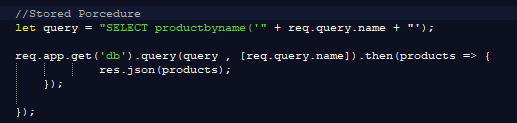
**The current stored procedure solves the sql injection problem :**

**I have inserted this script using PSQL into my pgguide database in postgres.**

**CREATE OR REPLACE FUNCCREATE OR REPLACE FUNCTION productByName(name VARCHAR(70)) RETURNS TABLE(id INTEGER, title VARCHAR(255), price NUMERIC) AS $$  
BEGIN  
 RETURN QUERY  
 SELECT products.id, products.title, products.price FROM products WHERE products.title LIKE name;  
END;  
$$ LANGUAGE plpgsql;  
query = "SELECT get\_products('" + req.query.name + "');TION productByName(name VARCHAR(70)) RETURNS TABLE(id INTEGER, title VARCHAR(255), price NUMERIC) AS $$  
BEGIN  
 RETURN QUERY  
 SELECT products.id, products.title, products.price FROM products WHERE products.title LIKE name;  
END;  
$$ LANGUAGE plpgsql;  
query = "SELECT get\_products('" + req.query.name + "');**

The following snippet of code is used inside a get request the same as previously. Except this time the query that is called uses data returned from the stored procedure. The reason stored procedures work to prevent sql injections is because they work like a param query. In sense.

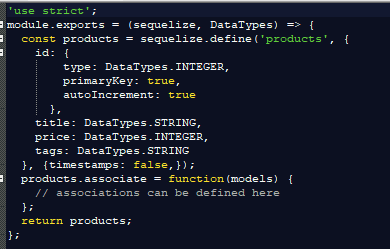
Since stored procedures are pre compiled and parsed before you run them , they just accept parameters that you feed in that will be processed inside the stored procedure, hence long sql statements will not work as they will not match the parameters the procedure is expecting. Preventing SQL injection. They work kind of like a function or method in a programming language. They processes parameters and if they are valid will return a result from the database.

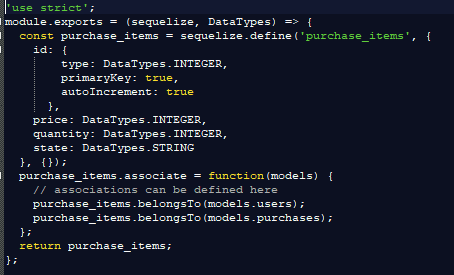


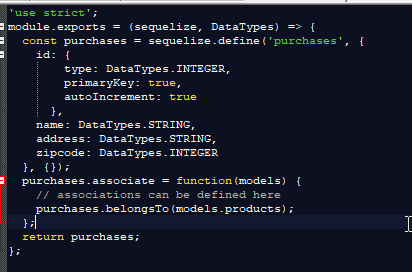
1. **Express and Sequelize ORM + Migrations:**

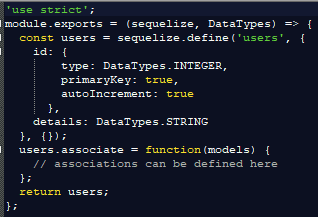
Here is included code from the models and the server. I did not include any of the boiler (Bootstrap) code that sequelize automatically generates such as the app directories, index.js and migration files. Can show them on request. I feel like they were not needed to show as they were automatic.

MODELS :

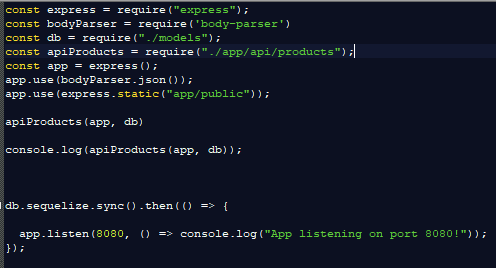




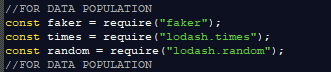


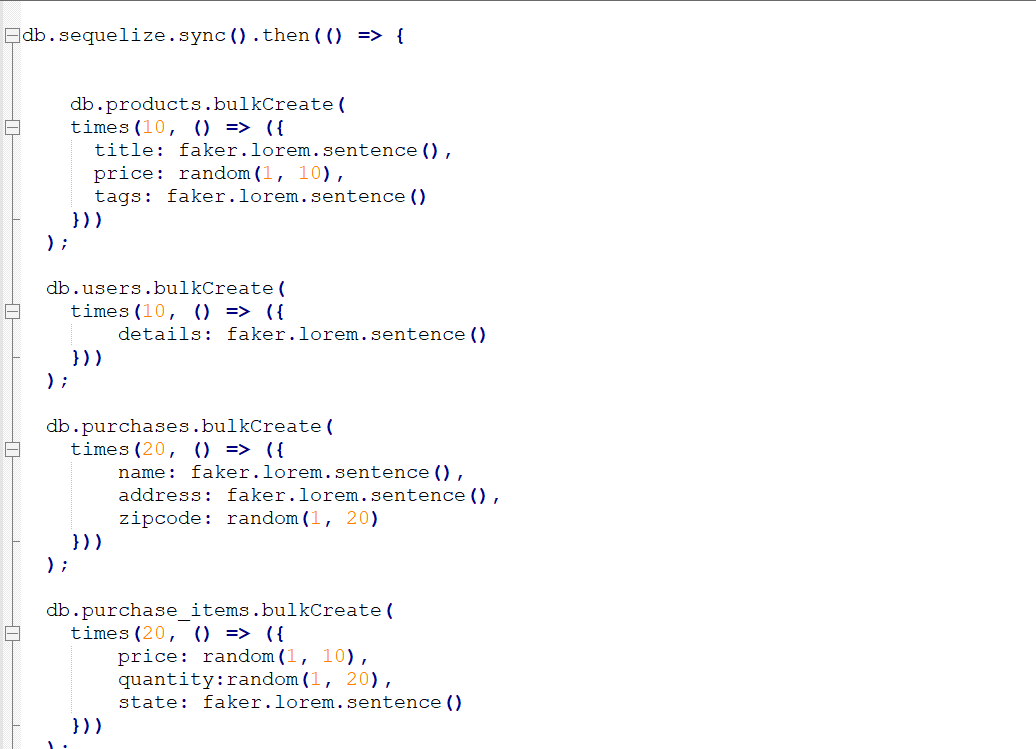


Server :



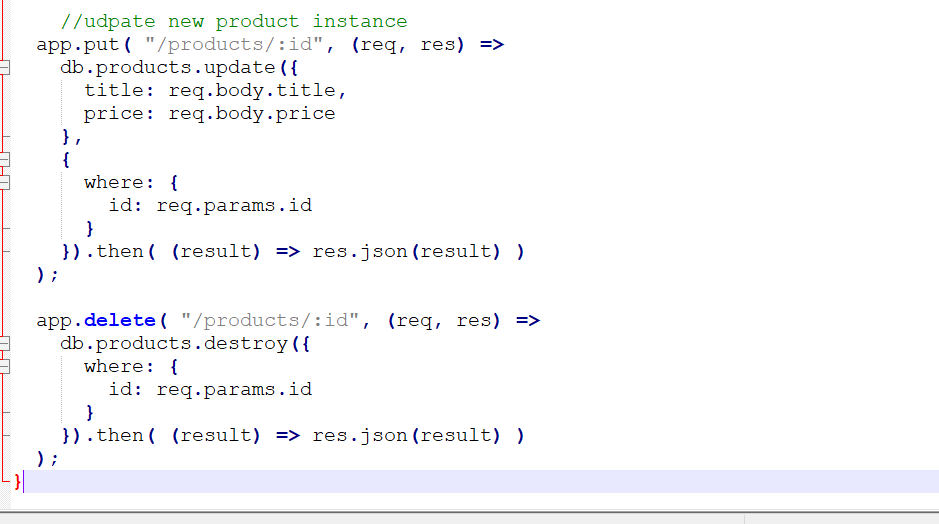
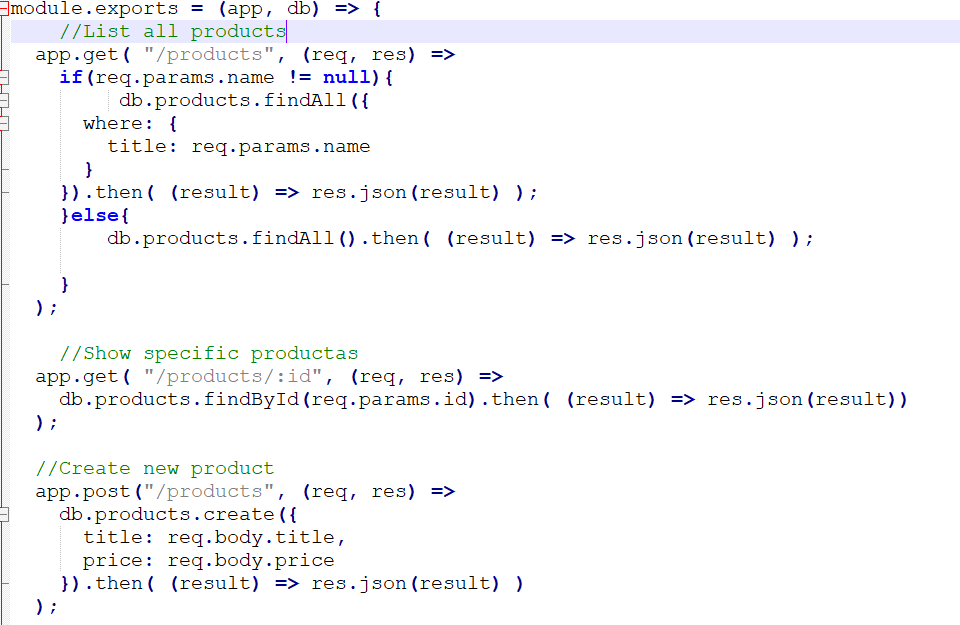
1. **Populate additional test data for all models :**



****

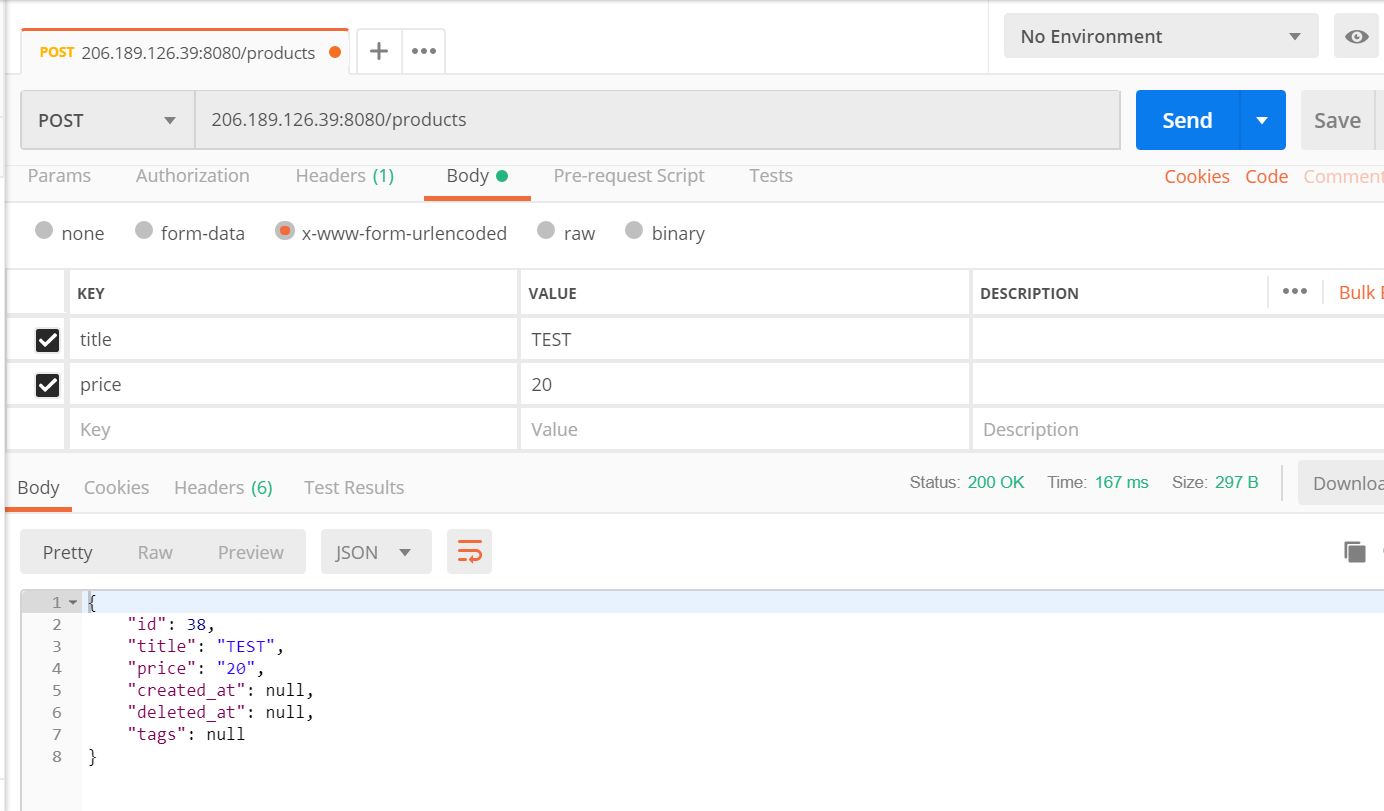
1. **RESTFul API using Sequelize and Express :**

**CRUD functional rest api**



**First 3 are just tested by going to : MYSERVERIP:PORT/products or /products/id or /products?name=string**

**I used postman to test post , put and delete to test the API using encoded urls**

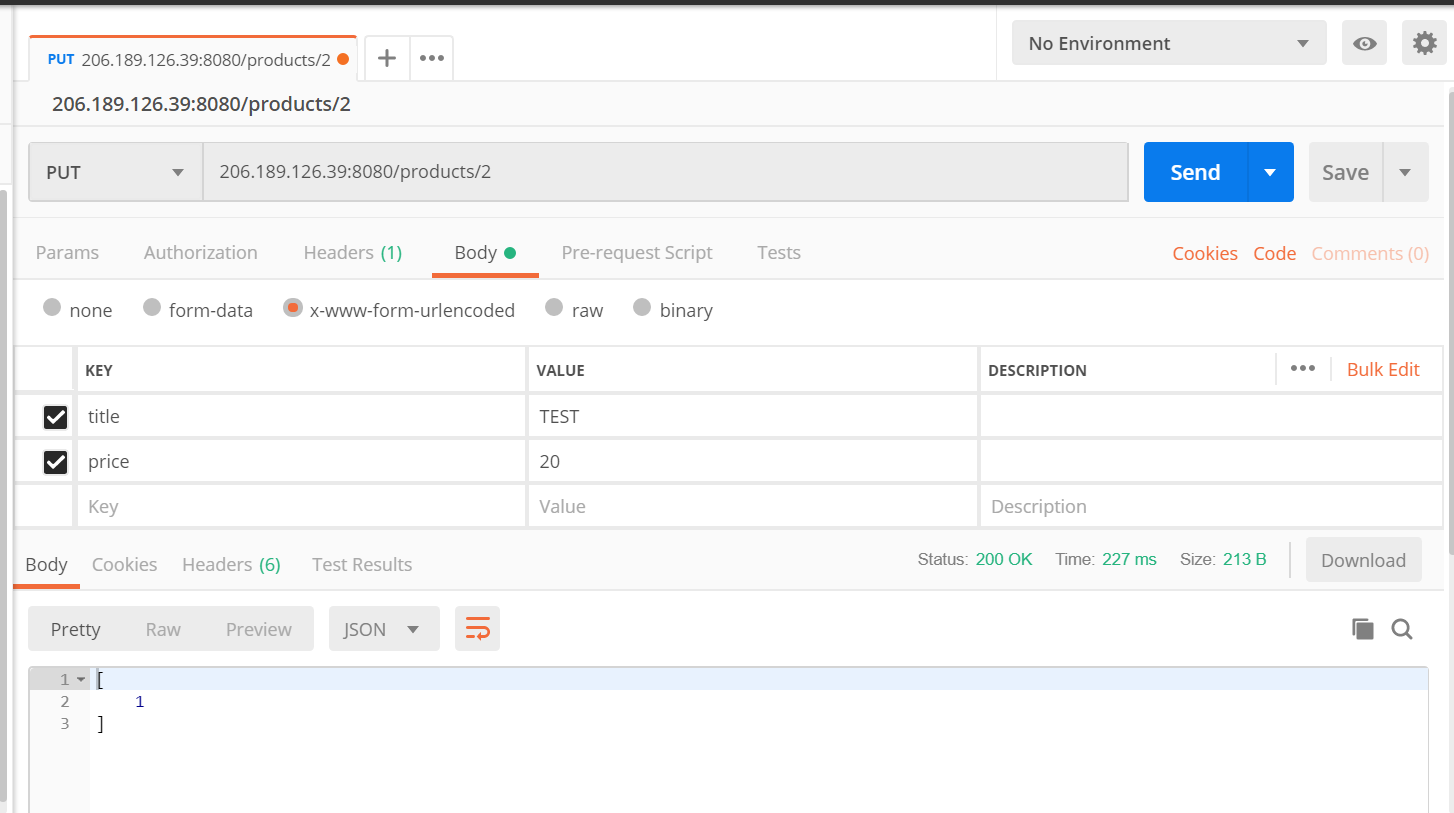
****

**After several errors and json body parser recieving empty body requests.**

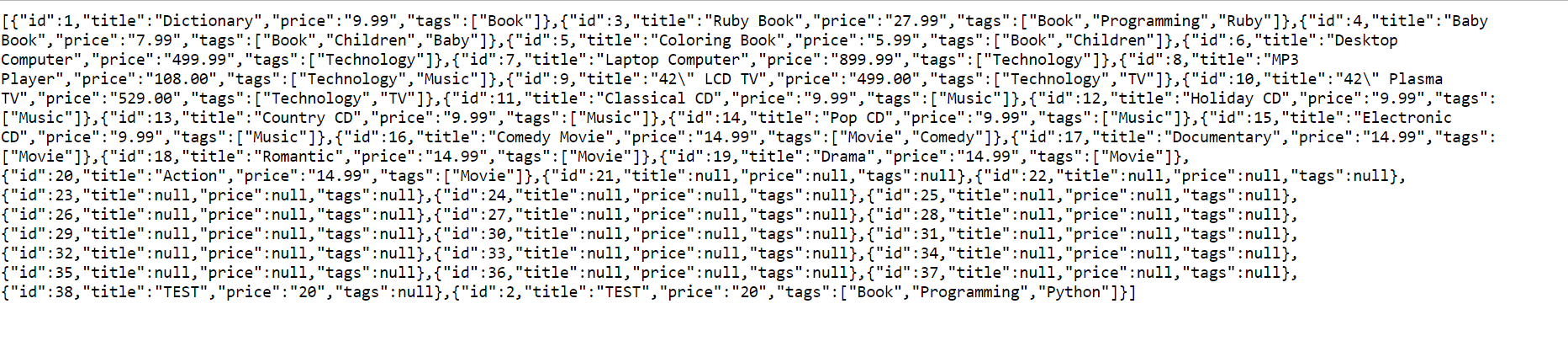
**I setup bodyparser to use encoded urls and the post request worked**

****

**Put request**

****

**As you can see in the last json object , the product with the id 2 as specified in the url is updated with the new details**

****