# 

Enterprise Application Development

Lab 1

DT228/4

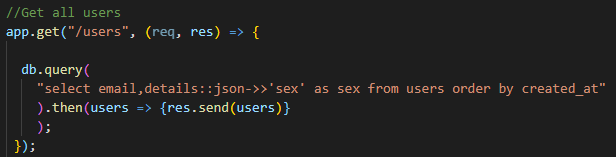
C15321906

Dylan Kieran

# Problem Sets

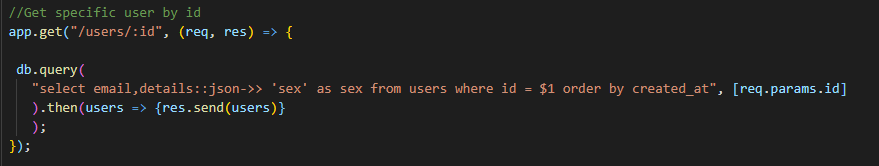
## 1.

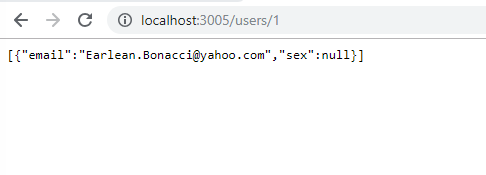
|  |  |
| --- | --- |
| GET /users | List all users email and sex in order of most recently created. Do not include password hash in your output |
| HTTP Request | http://localhost:3005/users |



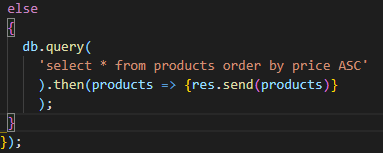


|  |  |
| --- | --- |
| GET /users/:id | Show above details of the specified user |
| HTTP Request | http://localhost:3005/users/1 |



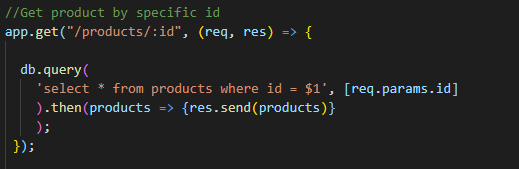


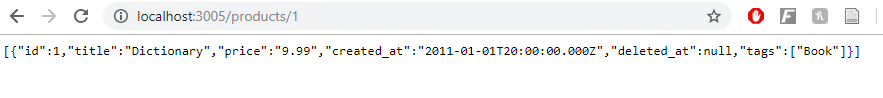
|  |  |
| --- | --- |
| GET /products | List all products in ascending order of price |
| HTTP Request | http://localhost:3005/products |



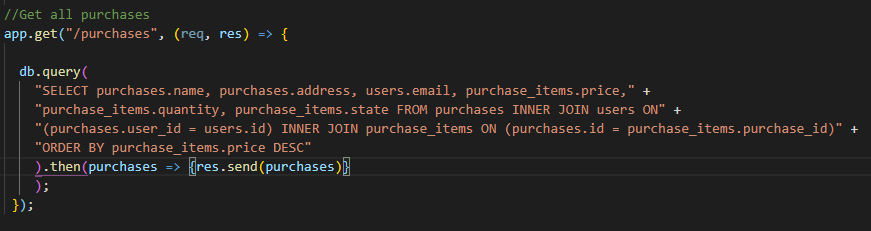
## C:\Users\dylan\OneDrive\Desktop\EnterpriseDev\8.png

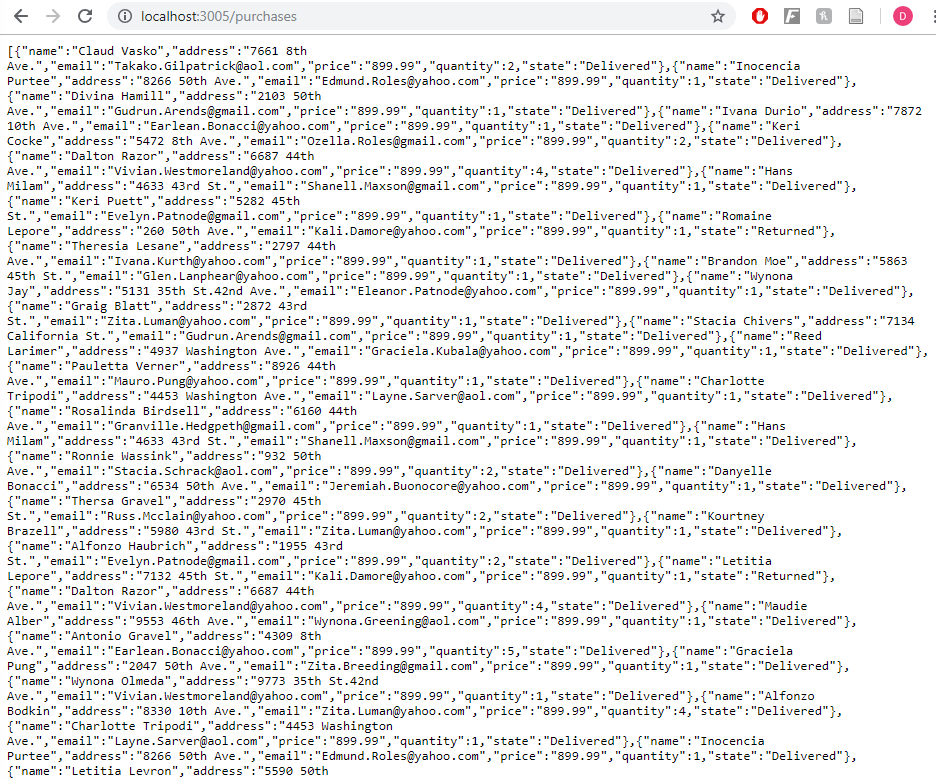
|  |  |
| --- | --- |
| GET /products/:id | Show details of the specified products |
| HTTP Request | http://localhost:3005/products/1 |





|  |  |
| --- | --- |
| GET /purchases | List purchase items to include the receiver’s name and, address, the purchaser’s email address and the price, quantity and delivery status of the purchased item. Order by price in descending order |
| HTTP Request | http://localhost:3005/purchases |



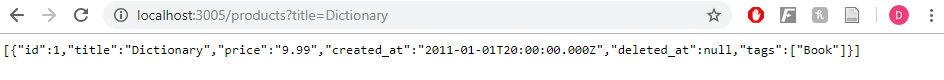


## 2.

|  |
| --- |
| GET /products[?name=*string*] |

*For your solution you should implement the query (badly) in such a way as to allow an attacker to inject arbitrary SQL code into the query execution. Show, using your badly implemented approach, how an attacker can craft a query string to allow the deletion of a product from the products table.*





In order to attack this solution an attacker would simply input the following http request into the browser.

[*http://localhost:3005/products?title=*](http://localhost:3005/products?title=)*Dictionary'; DELETE FROM purchase\_items WHERE product\_id = 1; DELETE FROM products WHERE title = 'Dictionary';*

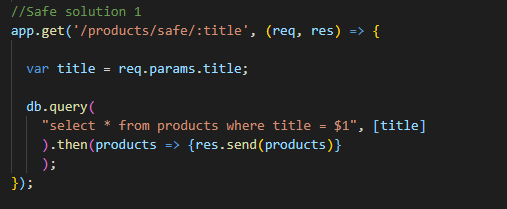
This would remove the foreign key dependencies and remove the dictionary book from the product table.

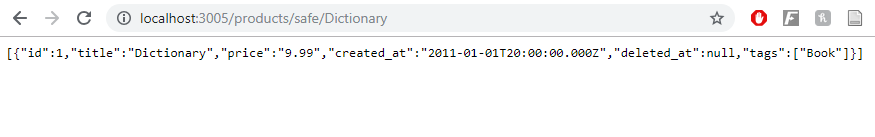
## 3.

|  |
| --- |
| Provide two solutions to eliminate the security hole in your approach from the previous section as follows:   * Using a parameterised query * Using a stored procedure using SQL or PLPGSQL whichever you prefer   Explicitly show that the injection attack is not now possible for each of your solutions  Again, you can just use MassiveJS as your database interface library here too. |

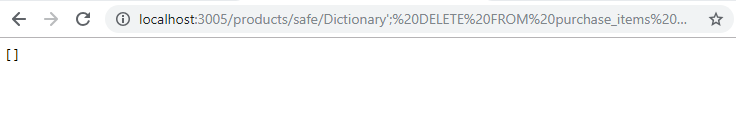
Injection Attack does not work against the following solutions:

## Parametrised Query

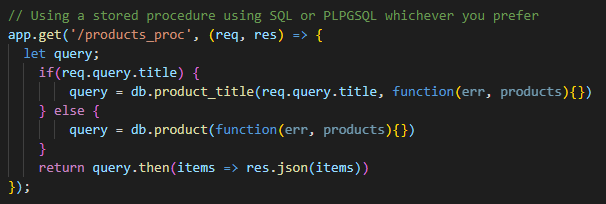


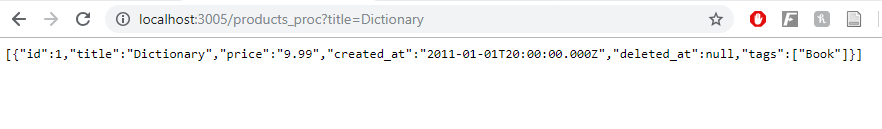


Injection:

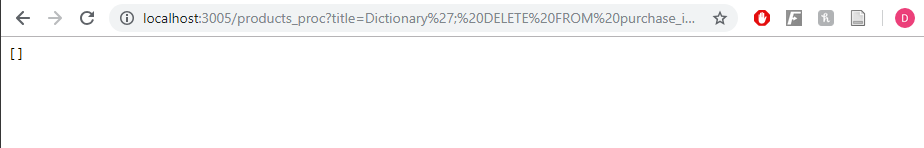


## Stored Procedure



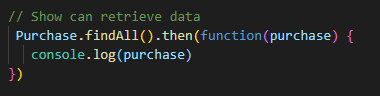


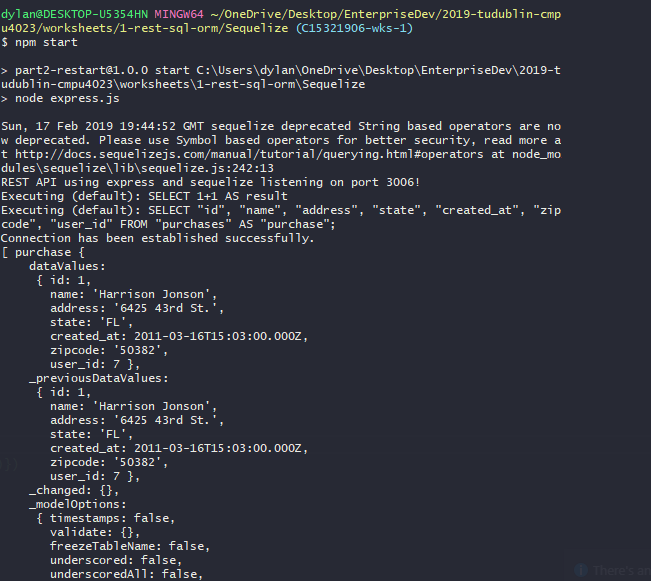
Injection:



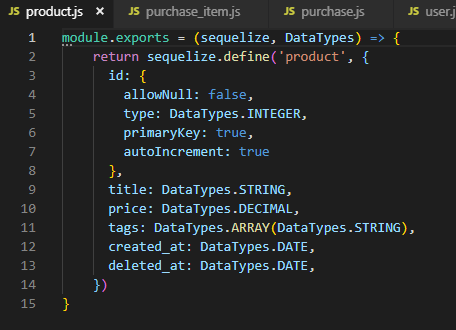
## 4.

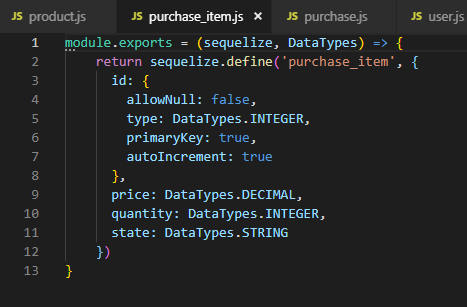
|  |
| --- |
| Create a brand-new Express project using the Sequelize ORM. Install and configure Sequelize and wire it up to the pgguide database. Verify that you have basic connectivity before proceeding.  Create Sequelize migrations for the pgguide sample database  Ensure that the appropriate associations and referential integrity checking are set up in your models |

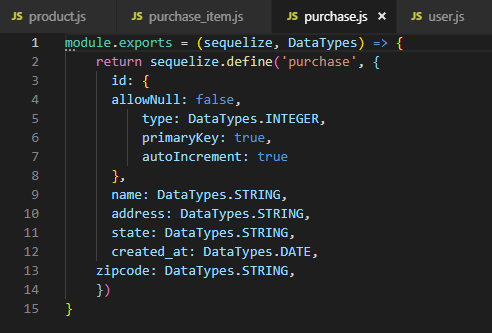


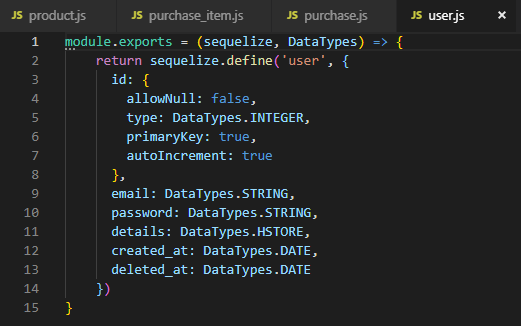


Migrations:

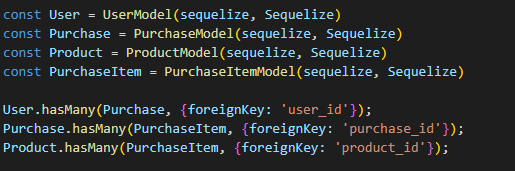






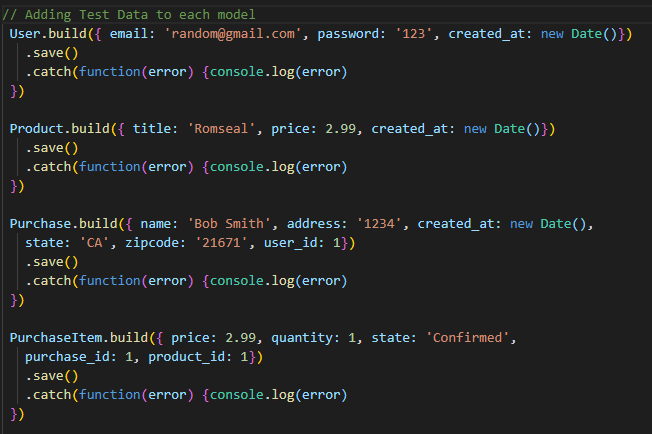


Referential Integrity:



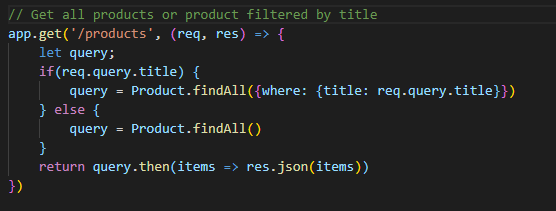
## 5.

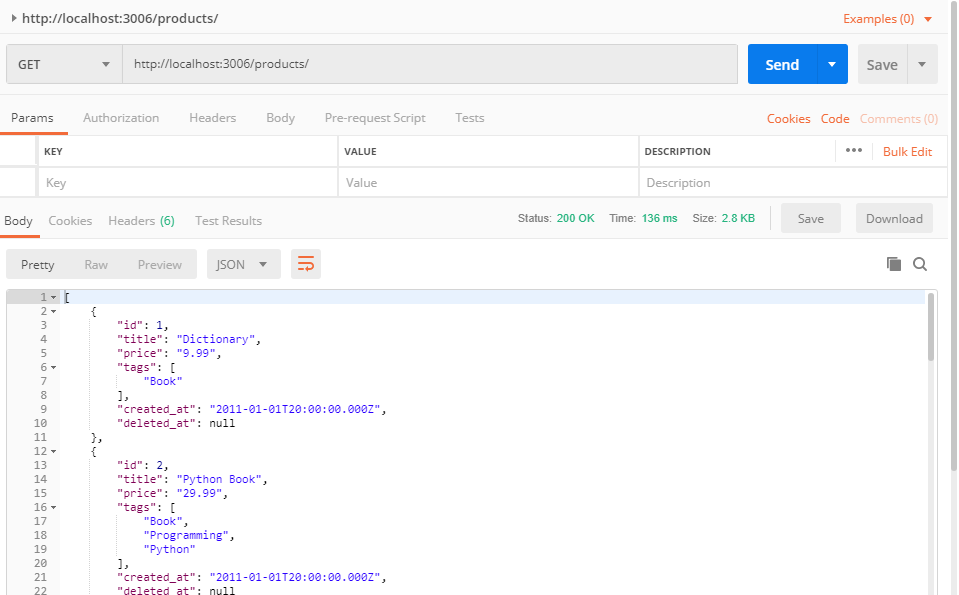
|  |
| --- |
| Use your models and JavaScript code to populate the database with some additional test data for all of the models above |

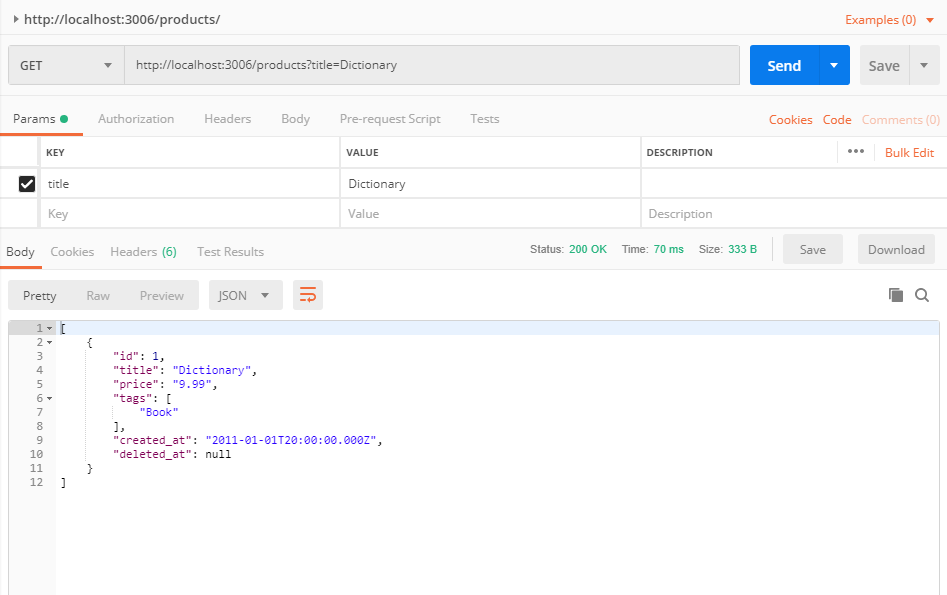


## 6.

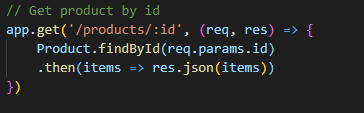
|  |  |
| --- | --- |
| GET /products[?name=string] | List all products |

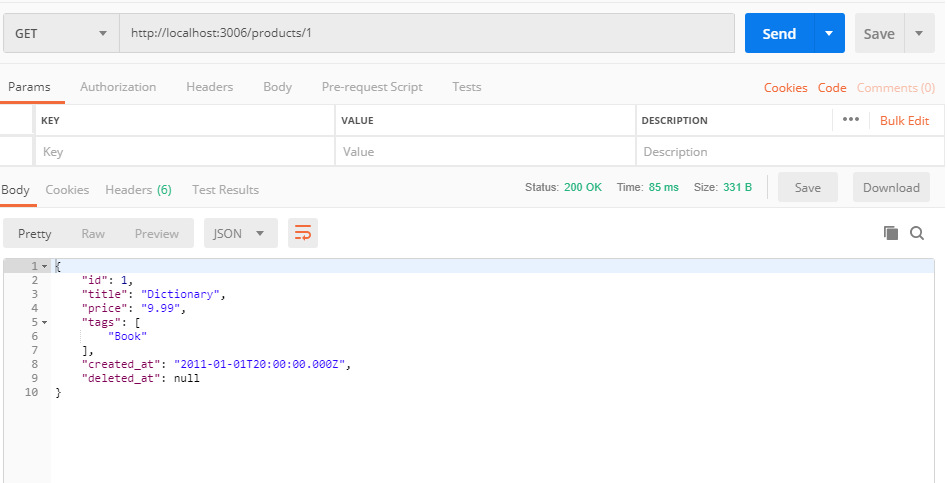




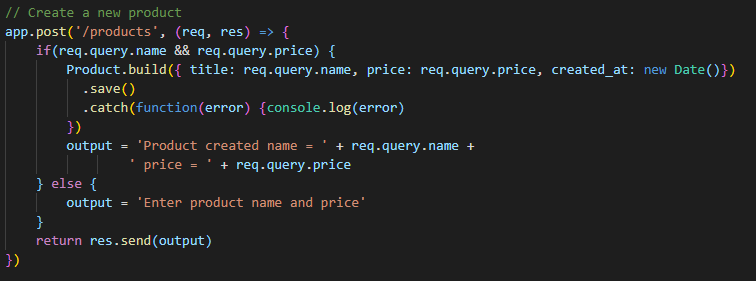


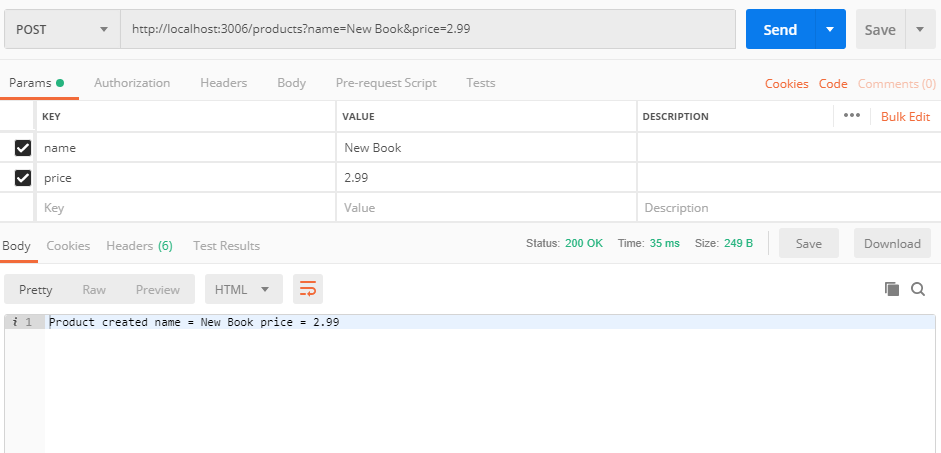
|  |  |
| --- | --- |
| GET /products/:id | Show details of the specified products |

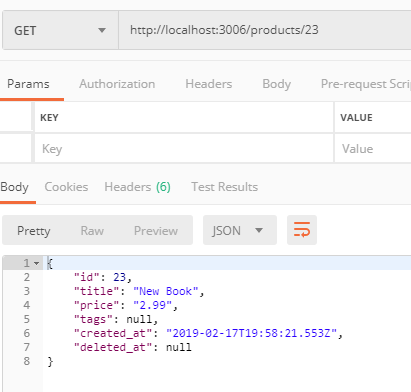




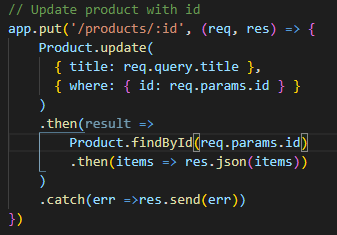
|  |  |
| --- | --- |
| POST /products | Create a new product instance |

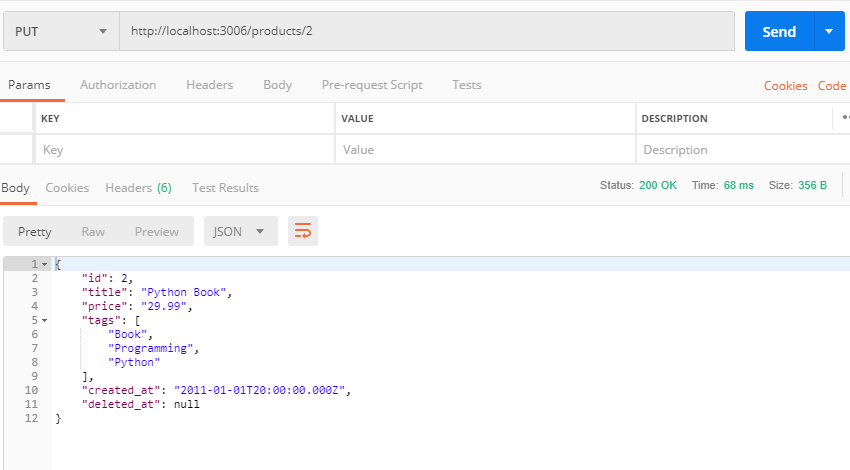






|  |  |
| --- | --- |
| PUT /products/:id | Update an existing product |





|  |  |
| --- | --- |
| DELETE /products/:id | Remove an existing product |

