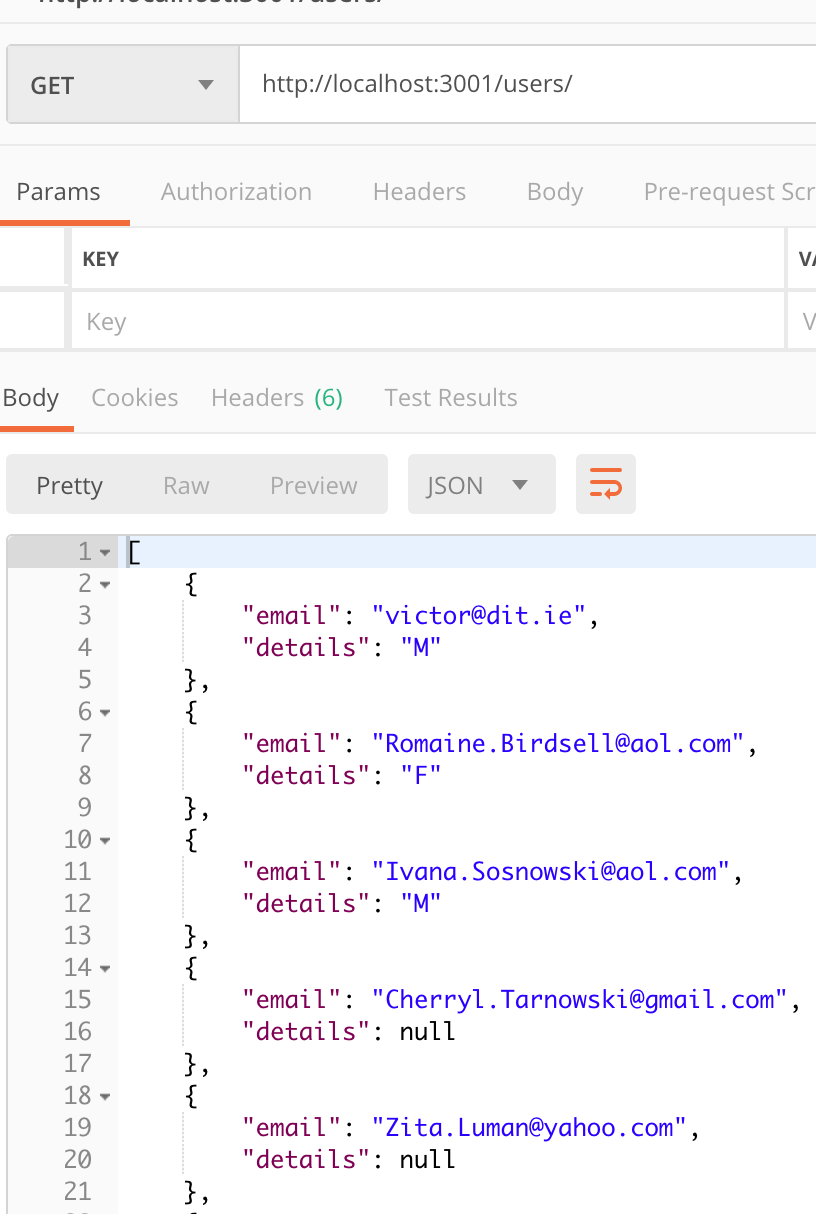
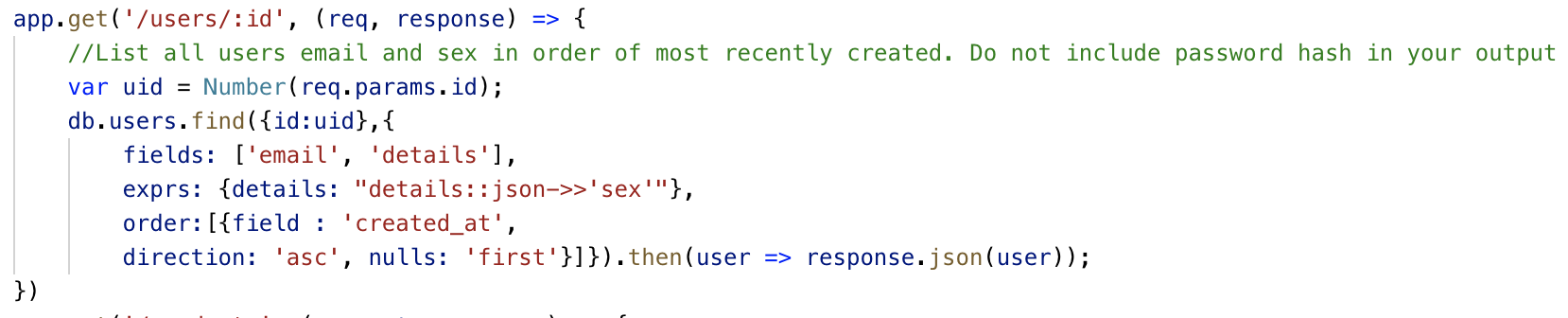
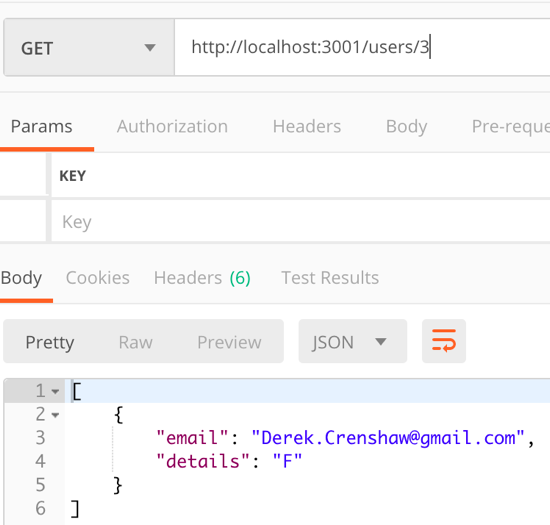
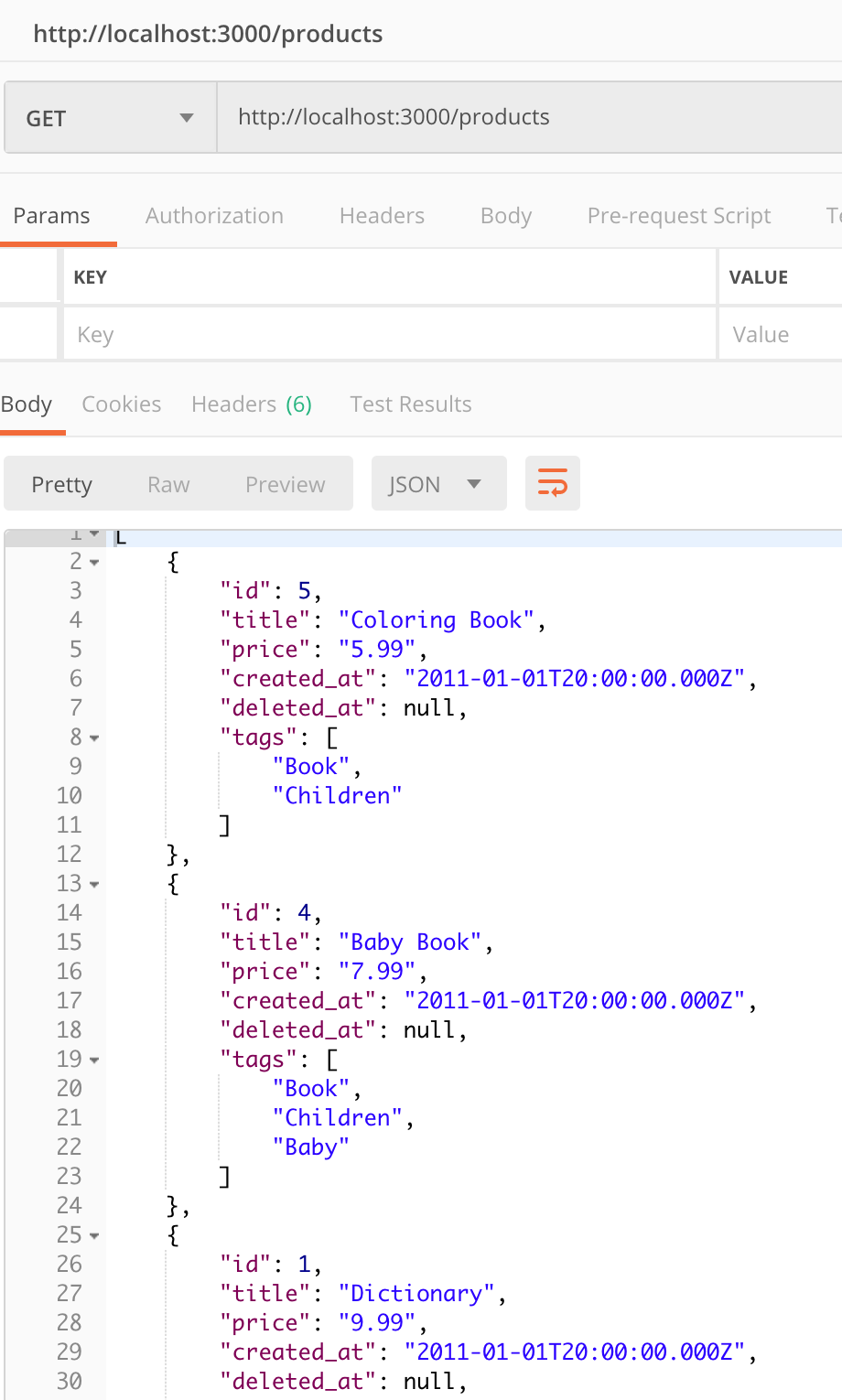
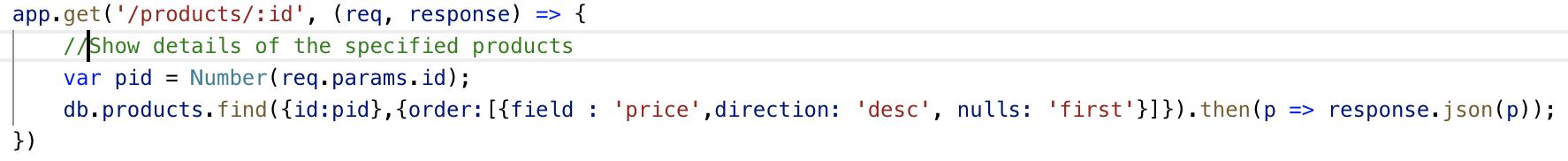
1. **Using Node, Express and Massive create the following HTTP API endpoints serving the following resources as JSON documents**



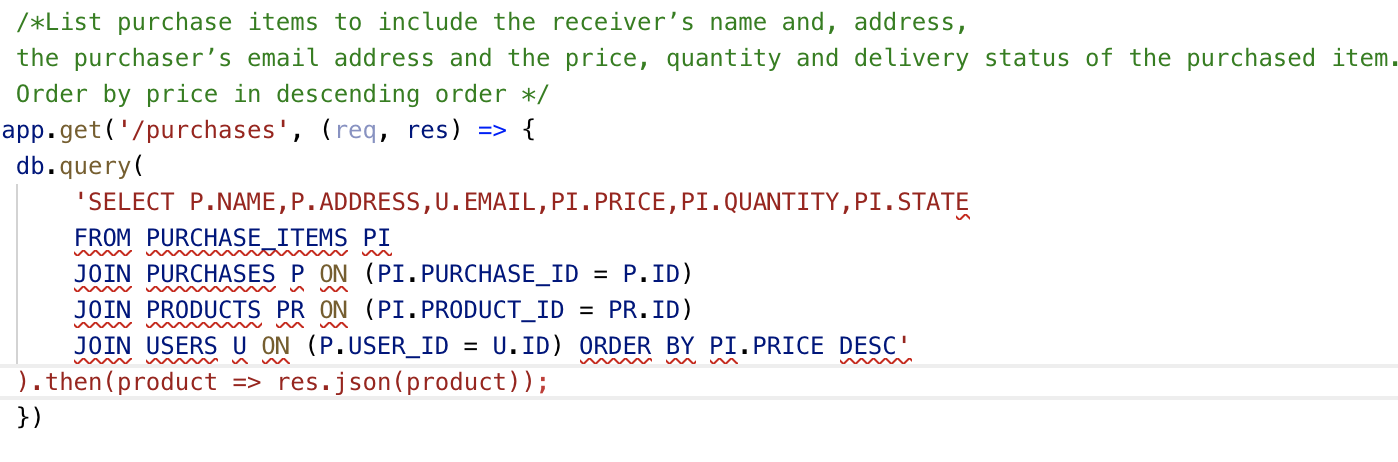


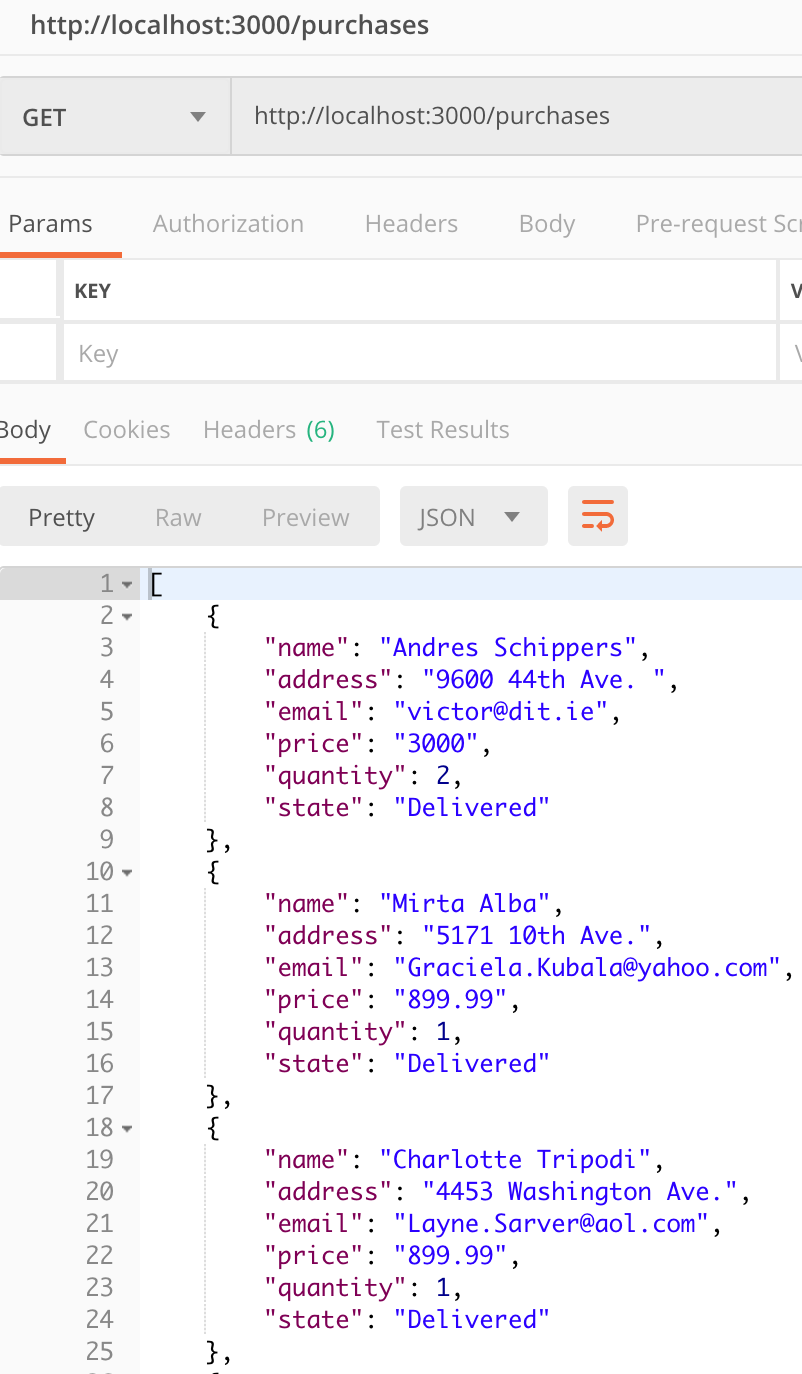




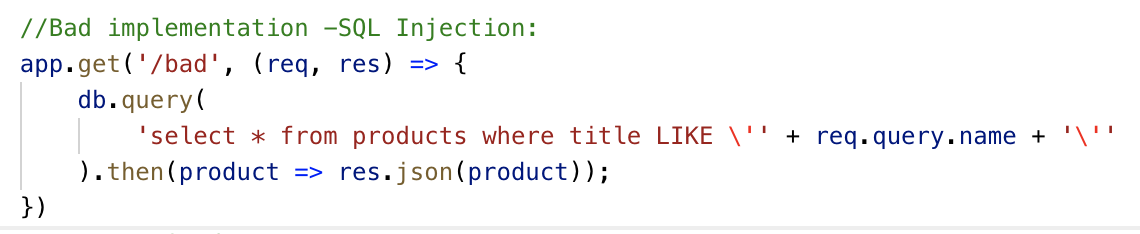








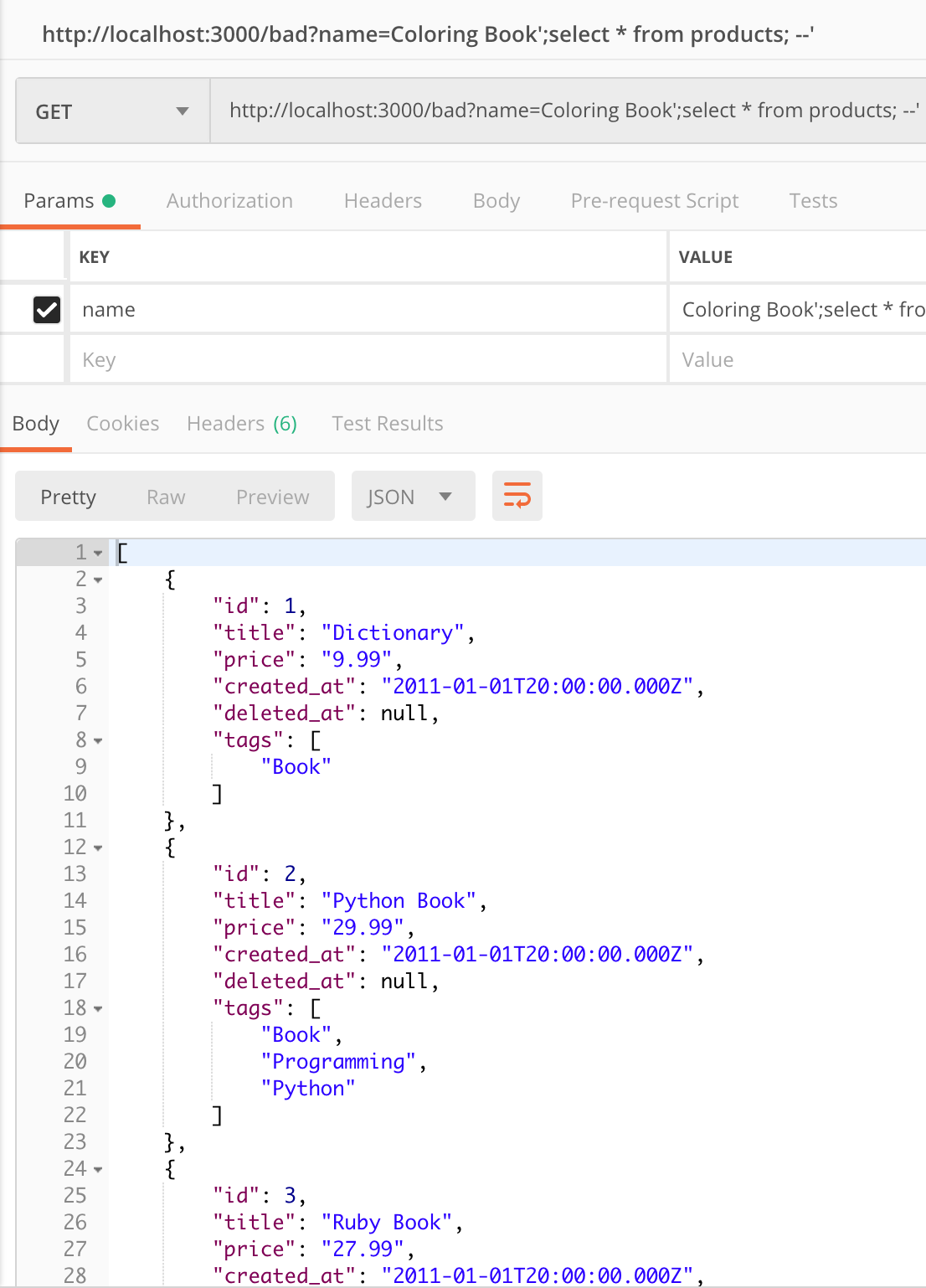
2. Building on your solution to part 1 for the API to the products resource from the pgguide database, extend the product indexing endpoint to allow the filtering of products by name as follows: GET /products[?name=*string*]



To test this bad implementation we inject SQL at the end of a regular query:

http://localhost:3006/bad?name=Coloring Book';drop table products; --' The following SQL injection doesn’t display the Coloring book product, instead it deletes the entire Products table from the database.

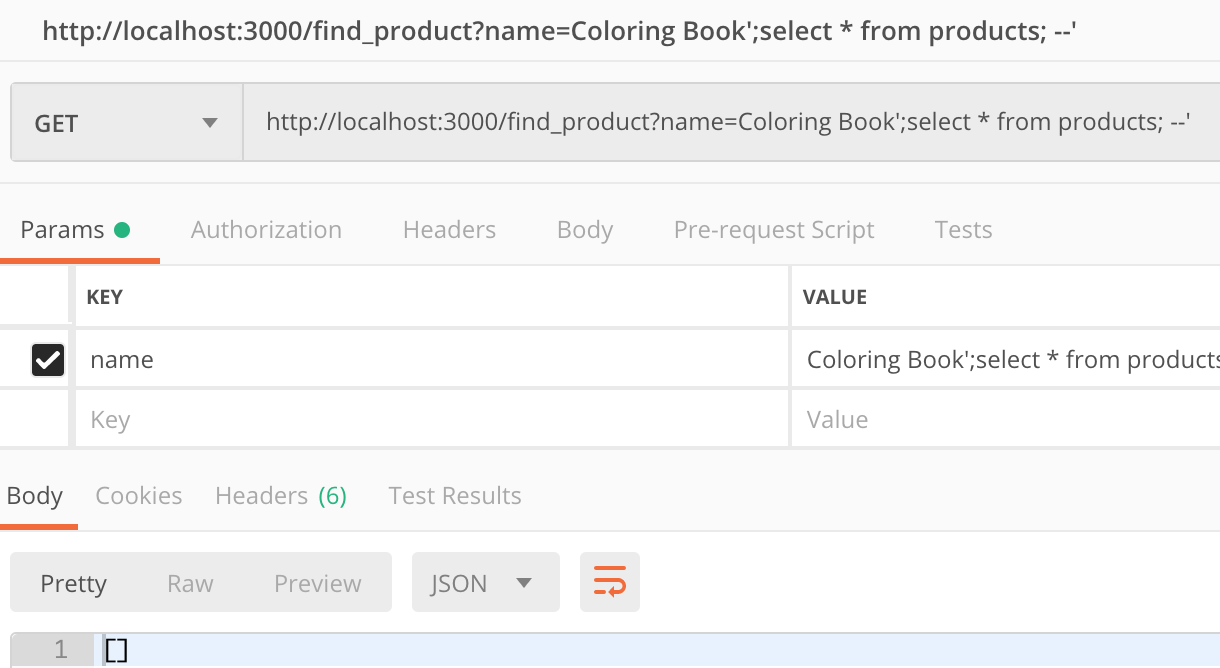
The following screenshot shows SQL injection in which all products of the database are displayed:



**3. Provide two solutions to eliminate the security hole in your approach from the previous section as follows:**

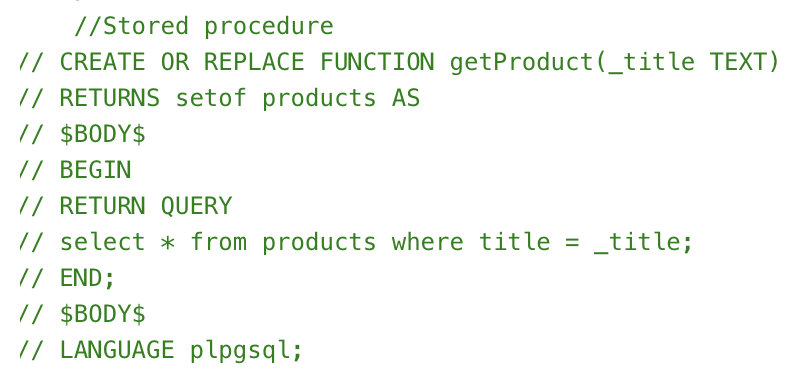
* **Using a parameterised query :** We can verify that SQL injection doesn’t work.

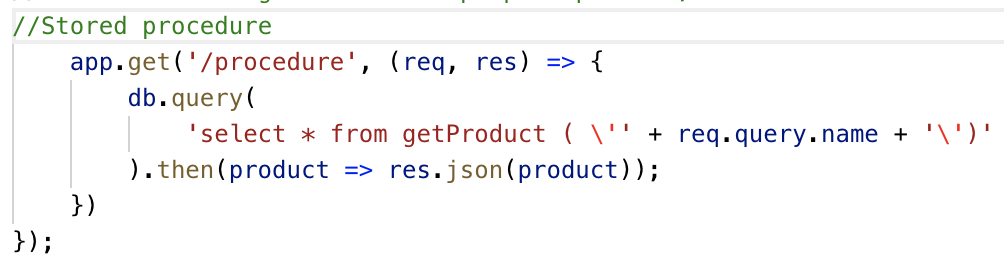
****

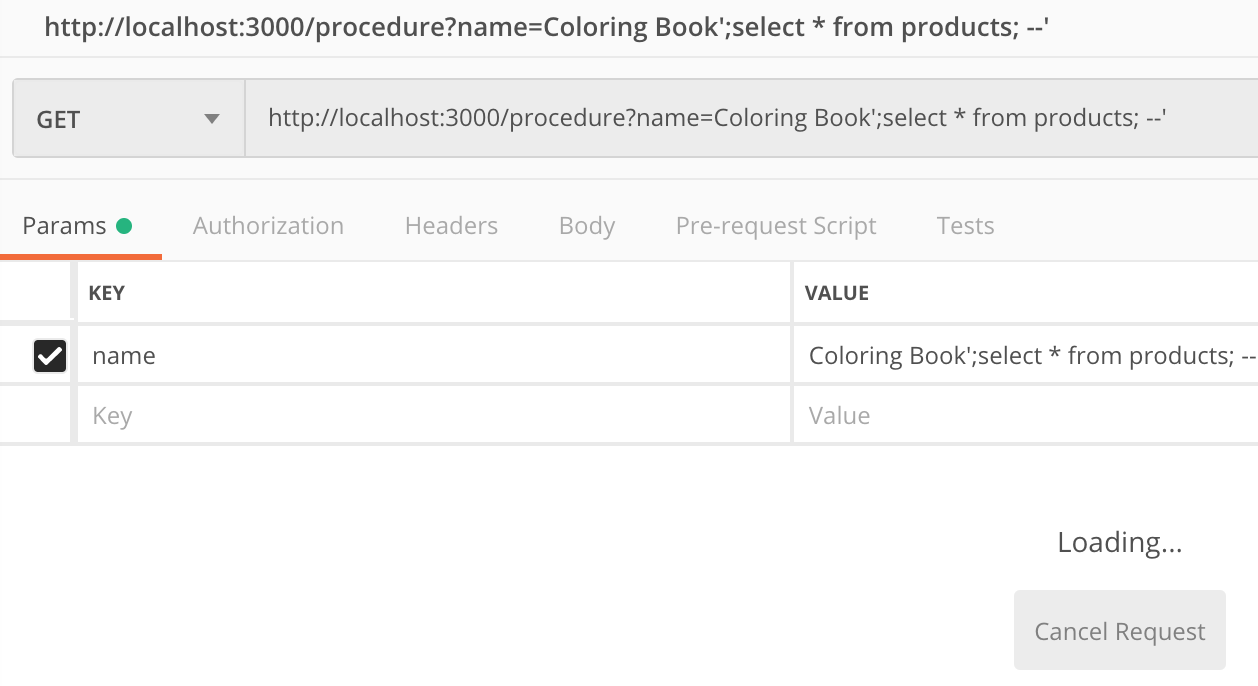
****

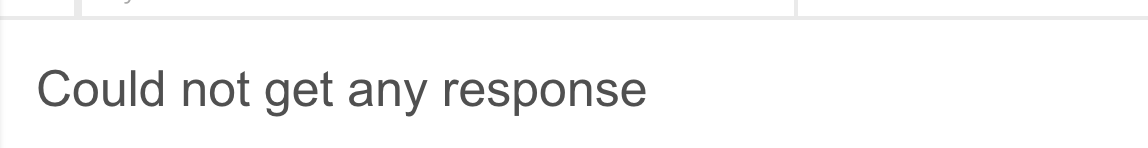
* **Using a stored procedure using SQL or PLPGSQL whichever you prefer**

The following code is executed on PostgresSQL Database. It creates a procedure/function that it can be queried and prevents SQL injection.

****

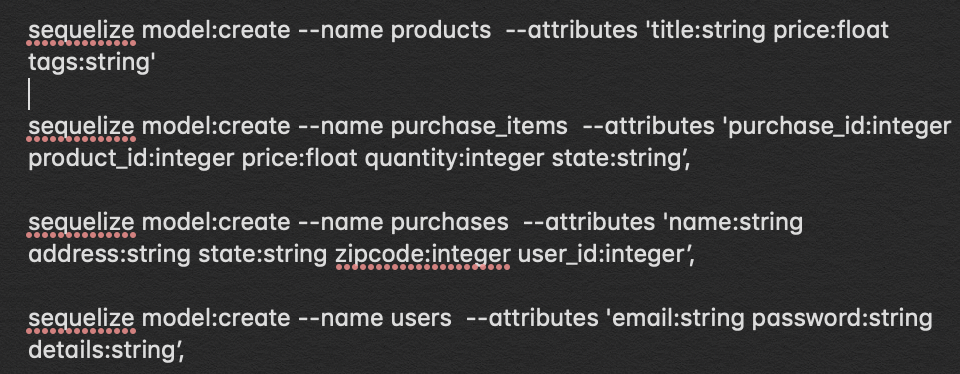




****

**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.**

Using the following commands a model representation of the existing table in the database were created:



This creates a file for each table of the database that looks as follows:

The appropriate associations and referential integrity of the models were also created.



We can easily populate the tables by using the models previously created.

