# Goal

The goal is to test a REST API with the URL

# Scope of testing

Functional testing will be done on the different functionalities offered by the API

For Products:

* Get a list of all products
* Get product by id
* Create a new product
* Update product by id

For Orders:

* Return a list of all order
* Create an order
* Return an order by id

Tests will verify that

* The API performs the intended actions.
* the API returns the expected data (validate data in the database with data send and received via the API).
* errors are handled (invalid requests, missing parameters, and errors).

# Risks and Limitations

Tests should include edge cases, such as invalid inputs or extreme values, both can result in unexpected behavior if not handled by an expected error.

My tests are only functional, but load testing should be done as well.

# Test Cases

The Url can be changed in the file globalConfig.py

* baseUrl = "https://tv.test.io/api/v1"

For local testing purposes I used mockAPI (https://designer.mocky.io/) to return the expected responses listed in the YAML file.

mock=True

mockUrlAllProducts = "https://run.mocky.io/v3/8fd435f9-df96-4b37-994b-1ee686d0e850"

mockUrlProductById = "https://run.mocky.io/v3/0bcc0f2d-e5e7-43a4-a815-fcb48da041cf"

mockUrlProductByIdNotFound = "https://run.mocky.io/v3/67df6260-c96b-4299-bab0-e3493d930dd1"

mockUrlCreateProduct = "https://run.mocky.io/v3/9e14a025-891e-45e1-94da-089bfc79993a"

mockUrlCreateProductWithError = "https://run.mocky.io/v3/a12dedcd-572a-4059-a3cb-d8d038221ab1"

mockUrlPutProduct = "https://run.mocky.io/v3/d21e9211-2778-4317-873d-dab89336d769"

mockUrlGetOrders = "https://run.mocky.io/v3/c773d9f1-35e6-4685-a8cb-673e3a0fde58"

mockUrlGetOrdersById = "https://run.mocky.io/v3/27acedd1-ced6-448b-9384-d86a67902e03"

mockUrlOrderByIdNotFound = "https://run.mocky.io/v3/67df6260-c96b-4299-bab0-e3493d930dd1"

mockUrlCreateOrder = "https://run.mocky.io/v3/99ee7fb8-7197-4922-9d05-e2728df04a66"

mockUrlCreateOrderWithError = "https://run.mocky.io/v3/a12dedcd-572a-4059-a3cb-d8d038221ab1"

and this could be turn off by changing the parameter mock to False, the tests will then use the baseUrl.

## Product test cases

All tests regarding products are in the python file **test\_product.py**

## API call to return list of all products

#### test\_get\_products()

No parameters

### Tests:

* # Verify status code
* # Verify content-type
* # Verify response structure (Assuming a product object with 'id', 'name', 'description' and 'price')
* # Verify number of products in database is the same as number of products returned
* # Verify response values against database for first element returned

A test to check every items id, description and price can be ran, but might be long, assuming that the products are returned by product ID, I just verify data for the first one.

## API call with product id to return a product

#### test\_get\_productById()

The product id parameter can be changed in UrlParameters.py

# product ID for URL

exisitingProductId = 11

exisitingProductIdPayload = '{id": 11}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (Assuming a product object with 'id', 'name', 'description' and 'price')

# Verify response values against database

#### test\_get\_productByIdNotFound()

The product id parameter can be changed in UrlParameters.py

unknownProductId = 100000000000000000000

unknownProductIdPayload = '{id": 100000000000000000000}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "ID not found")

## API call to create a product

#### test\_post\_product()

The product id parameter can be changed in UrlParameters.py

#product creation

newProduct = {"name": "TSHIRT","description": "Product description","price": 10}

product\_name = "TSHIRT"

product\_description = "Product description"

product\_price = 10

### Tests:

* # Verify status code
* # Verify content-type
* # Verify response structure (Assuming a product object with 'id', 'name', 'description' and 'price')
* # Verify response values against the json passed
* # Verify response values against the database -> new product is inserted in database

#### test\_post\_product\_with\_errors(json\_data)

The test is called with several parameters to trigger errors

# The test will run with each of the parameters for the value json\_data

@pytest.mark.parametrize("json\_data", [UrlParameters.product\_error\_list1, UrlParameters.product\_error\_list2, UrlParameters.product\_error\_list3,

                                       UrlParameters.product\_error\_list4, UrlParameters.product\_error\_list5])

Values can be changed in UrlParameters.py

#product creation list with different errors

product\_error\_list1 = '{"name":  1,"description": "product name integer","price":  1}'

product\_error\_list2 = '{"name":  "ERRORDESC" , "description": 1, "price":  1}'

product\_error\_list3 = '{"name":  "ERRORPRICE", "description": "price char", "price":  "PRICE"}'

product\_error\_list4 = '{"name":  "aaaa", "description": "product name not uppecase", "price":  1}'

product\_error\_list5 = '{"name":  "1111", "description": "product name contains number", "price":  1}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "Invalid input")

## API call to update a product

#### test\_put\_product()

The product parameters can be changed in UrlParameters.py

#product update

updated\_product = {"name": "TSHIRTS","description": "New description","price": 15}

updated\_product\_id = 10

updated\_product\_name = "TSHIRTS"

updated\_product\_description = "New description"

updated\_product\_price = 15

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (Assuming a product object with 'id', 'name', 'description' and 'price')

# Verify response values against the json passed

# Verify response values against the database -> product is updated in database

#### test\_put\_productByIdNotFound()

The product parameters can be changed in UrlParameters.py

unknownProductId = 100000000000000000000

unknownProductIdPayload = '{id": 100000000000000000000}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "ID not found")

#### test\_put\_product\_with\_error()

# The test will run with each of the parameters for the value json\_data

@pytest.mark.parametrize("json\_data", [UrlParameters.product\_error\_list1, UrlParameters.product\_error\_list2, UrlParameters.product\_error\_list3,

UrlParameters.product\_error\_list4, UrlParameters.product\_error\_list5])

Values can be changed in UrlParameters.py

#product creation list with different errors

product\_error\_list1 = '{"name":  1,"description": "product name integer","price":  1}'

product\_error\_list2 = '{"name":  "ERRORDESC" , "description": 1, "price":  1}'

product\_error\_list3 = '{"name":  "ERRORPRICE", "description": "price char", "price":  "PRICE"}'

product\_error\_list4 = '{"name":  "aaaa", "description": "product name not uppecase", "price":  1}'

product\_error\_list5 = '{"name":  "1111", "description": "product name contains number", "price":  1}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "Invalid input")

## Order test cases

All tests regarding orders are in the python file **test\_order.py**

## API call to return list of all orders

#### test\_get\_orders()

No parameters

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (Assuming a product object with 'id', 'time', 'name', 'items', 'name' and 'quantity' in items and 'subtotal')

# Verify number of orders in database is the same as number of orders returned

## API call with order id to return an order

#### test\_get\_orderById ()

The order id parameter can be changed in UrlParameters.py

# order ID for URL

exisitingOrderId = 10000

exisitingPOrderIdPayload = '{id": 10000}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (Assuming an order object with 'id', 'name', 'time', 'subtotal')

# Verify response values against database table order

# Verify response values against database table order\_products and products

#### test\_get\_orderByIdNotFound ()

The order id parameter can be changed in UrlParameters.py

unknownOrderId = 100000000000000000000

unknownOrderIdPayload = '{id": 100000000000000000000}'

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "ID not found")

## API call to create an order

#### test\_post\_order()

The order parameters can be changed in UrlParameters.py

#order creation

newOrder = {"name": "John Doe","items": [{"productId": 111,"quantity": 1}]}

expectedNewOrderCReatedItem = {"items": [{"name": "TSHIRT","quantity": 1}]}

order\_name = "John Doe"

item\_productId = 11

item\_quantity = 1

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (Assuming an order object with 'id', 'name', 'time', 'subtotal')

# Verify items passed are correctly returned

# Verify response values against database table order

# Verify response values against database table order\_products and products

#### test\_post\_order\_with\_errors(json\_data)

The test is called with several parameters to trigger errors

# The test will run with each of the parameters for the value json\_data

@pytest.mark.parametrize("json\_data", [UrlParameters.order\_error\_list1, UrlParameters.order\_error\_list2, UrlParameters.order\_error\_list3])

### Tests:

# Verify status code

# Verify content-type

# Verify response structure (message)

# Verify response message (Assuming an error with message "Invalid input")

# Results/Feedback and Considerations

## Creation of Application tables

I used SQL server locally and changed the create scripts :

CREATE TABLE products (

id bigint IDENTITY(1,1),

name text,

description varchar(30),

price numeric

);

CREATE TABLE orders (

id bigint IDENTITY(1,1),

name text NOT NULL,

created\_at datetime default CURRENT\_TIMESTAMP

);

CREATE TABLE order\_products (

id bigint IDENTITY(1,1),

order\_id bigint NOT NULL,

product\_id bigint NOT NULL,

quantity int NOT NULL,

price int NOT NULL

);

Considerations :

The table product does not have NOT NULL for name, a product with a blank name could be created

The table order\_products should not have INT for price but numeric the same way as in the table orders

## API Authorization key

No authorization key was mentioned in the YAML

Considerations :

If an authorization key was needed, I would create test to verify the behavior, and all other tests should include the header with correct auth-key

Considerations :

I would add a response code **400 Bad request** in case an id is not passed: the URL is ending with “/product/” with the message “product ID is missing”

I would add a response code **422 Unprocessable Content** in case the ID is not a valid integer with the message “product ID is not an integer”

## API call to create a product (POST)

Considerations :

I would add response code in case a product with a similar name exists and create a test to validate the functionality

## API call to get orders (GET)

Considerations :

A test could be written to check the data of the first order returned in the list of orders (same way I did with product) and assuming that orders are returned by order.id

## API call to create an order

Considerations :

I would add a response code **400 Bad request** in case an id is not passed: the URL is ending with “/order/“ with the message “order ID is missing”

I would add a response code **422 Unprocessable Content** in case the ID is not a valid integer with the message “order ID is not an integer”

A test to check the limit of items that can be added in a single post order call could be written