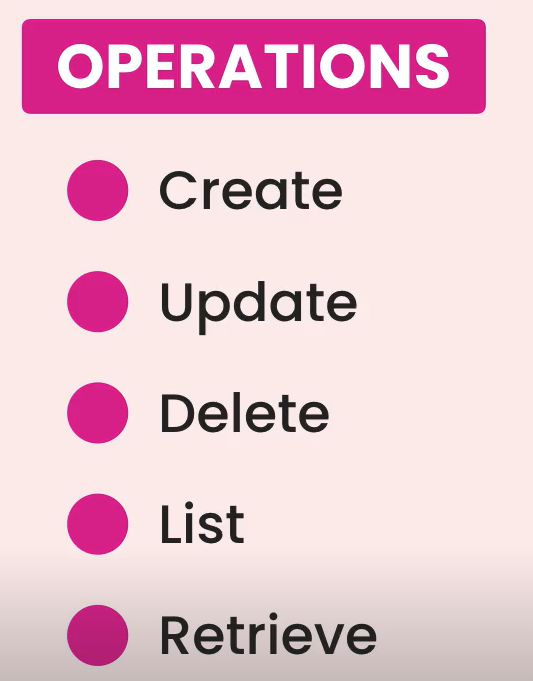




**What is Automated testing** :

Over the last few sections we have built a few end points for managing collections, products, carts and orders.

Now each of these endpoints support various operations.



Some of these operations involve a bunch of rules. For example only admin users are allowed to modify products.



So if an anonymous user or someone who is not an admin tries to modify a product, they should get an error.

Now here is the thing, we can manually test these endpoints in the browser, but as our application gets more complex, the time we need to manually test these functions increases exponentially.



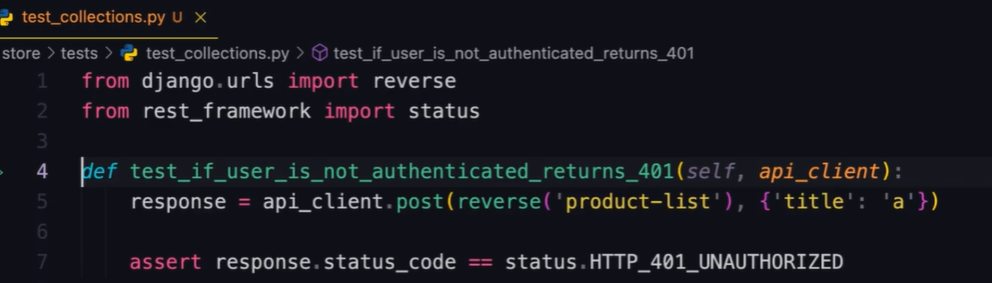
Plus over time we are going to forget various rules hidden behind these endpoints, unless we document them in a scripts that a tester can use to test various scenarios.

This is where *automated testing* comes to rescue,

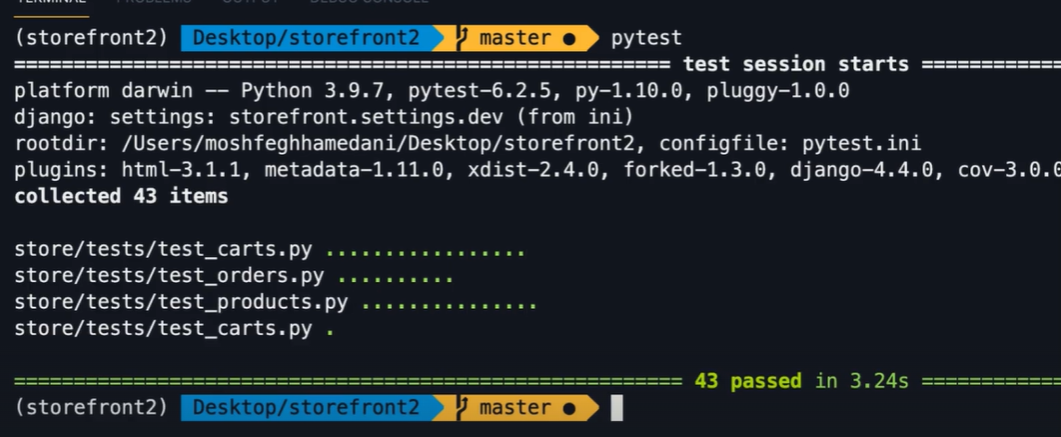


With automated testing we can write code to test our endpoints and their business rules.

So we write that code once and run it over and over.



Every time, we change our software or every time we want to deploy it, we run hundreds or thousands of test within seconds and see if we have accidently broken something or not.



So automated testing allow us to write better code and release it with more confidence.

**Test Behaviors, Not Implementation**:

***“****One of the reasons a lot of people fail with automated testing is that they test implementation, not behaviors****”***.

For example, let’s say we want to test a microwave. We press the start button and observe the monitor. We expect the monitor to show a timer running for one minute. So this is how we expect a microwave to behave.

This is how we test it, we don’t open up a microwave and test the electrical signals going in and out of every transistor.

Testing software is same.



We should test how the software behaves not how its implemented. Because the implementation may change over time.

A lot of people test building blocks of an API like models, views, routers, serializers and so on…



Over time our implementation may change, we may replace a function based view with a class based generic view.

Similarly we may split our model into two different models or combine two models into one.

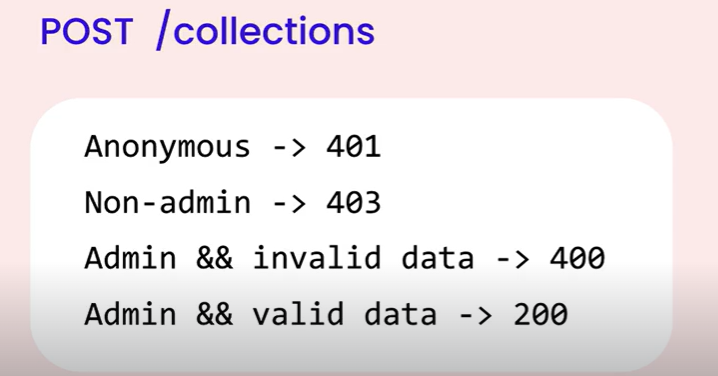
These are implementation details and they are prone to change.

*If we write tests for these details, our tests will break as we change the implementation and then we have to spend a lot of time fixing or rewriting these tests*.

In simple words, ***we should test how an API behaves not how its implemented***.

For example, Let’s say we want to test the creation of a *collection*. Here we send a POST request to the collections endpoint.

Now how does this endpoint behave? That’s what we need to test. Here we have a few scenarios.



🡪 If the client is not authenticated we expect a **401** (*unauthorized*) response.

🡪 If the client is authenticated but the current user is not an admin, then we expect a **403** (*forbidden*)response.

🡪 If the current user is an admin but the request does not include a collection name, then we expect 400 (*bad request*) and also the request body should include an error message for the collection name.

🡪 If the current user is an admin and our request includes a collection name we expect 200 *OK* response and the id of the new collection should be included in the response body.

So this is how our collections endpoint should behave when it comes to creating a collection. We can test these behaviors manually by hand in the browser or we can automate it using code.

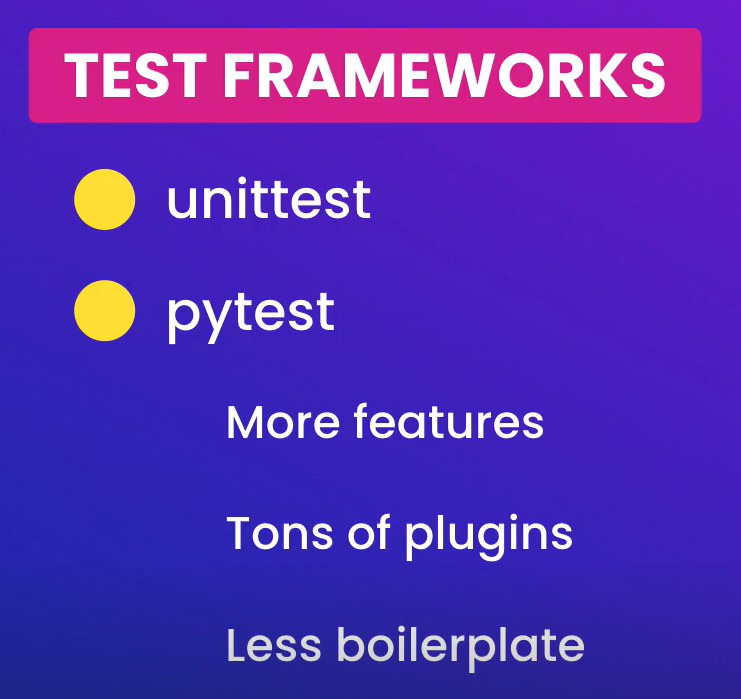
So let’s emphasize it one more time,



**Tooling**:

Just like we need a framework to create an API, we also need a framework for writing tests.

A test framework gives us a structure for writing tests as well as a program to run our tests and give us a report.



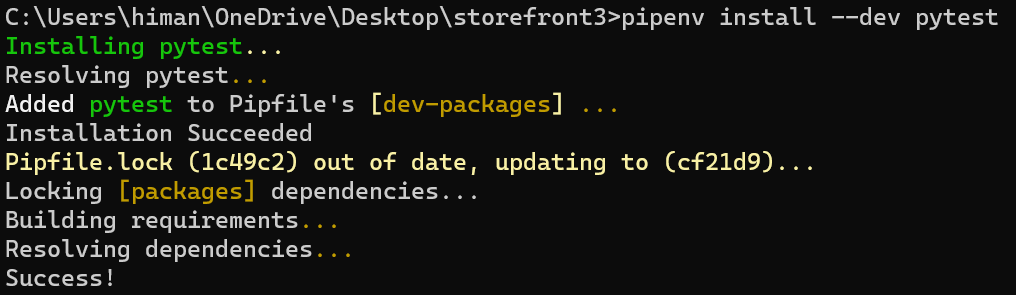
Here is side by side comparison of the same test written with *unittest* and *pytest*,



We can obviously see that tests written with pytest are shorter, cleaner and more concise.

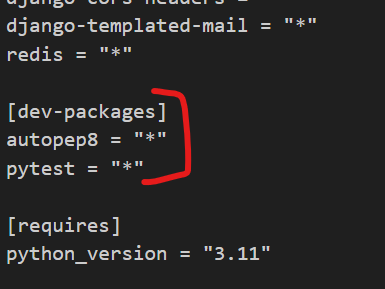
So in the terminal we are going to install *pytest*,

pipenv install --dev pytest



Note: Here --*dev* means development. So we are going to install pytest as a development dependency hence it’s not a dependency that we are going to deploy with our application to work (*only for development not deployment*).

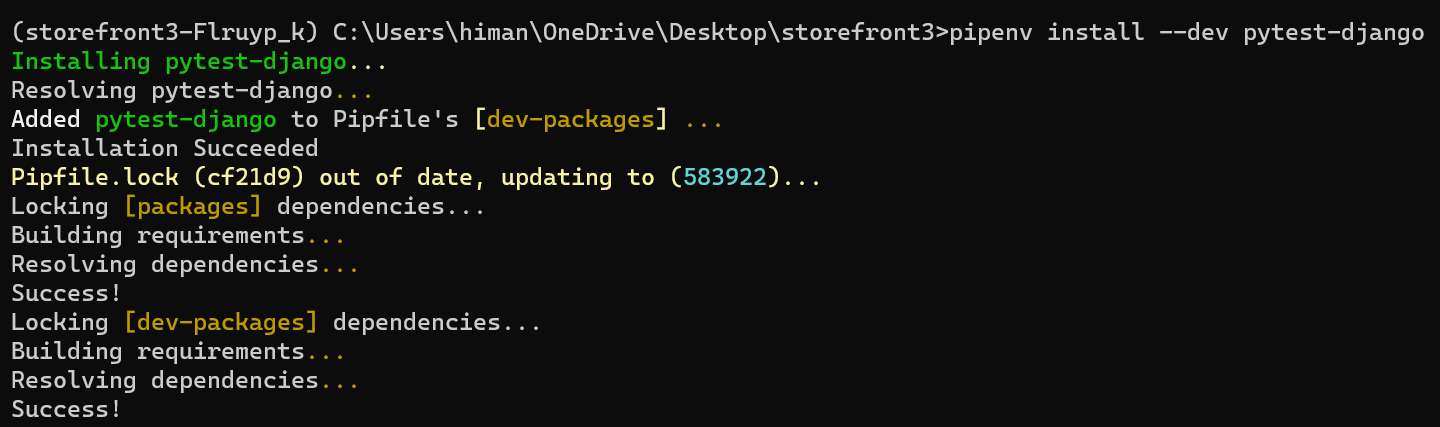
If we go to our *pipfile*,



We have our dev – packages. Packages that we use for development.

Now to test Django applications with pytest, we should also install one of the pytest plugins for django.

pipenv install --dev pytest-django



Our dev-packages are also updated,

[dev-packages]

autopep8 = "\*"

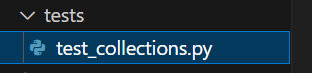
pytest = "\*"

pytest-django = "\*"

**Your First Test**:

We will go in the store app and create a new folder called ***tests*** (*name of the folder is important because that’s what pytest looks for*).

In this folder we are going to add a new file and its name should start from ***test\_*** . Since we are going to test collections endpoint so we name it ***test\_collections.py***.



Now to define a test we create a function and its name should start from ***test\_*** (*This is a convention which we should stick with for pytest*).

It’s really important that our test function identifies clearly what behavior we are testing.

Let’s say a scenario we are testing is that if a user is not authenticated we should get a 401 error.

def test\_if\_user\_is\_anonymous\_returns\_401():

Now the test name clearly identifies what behavior we are testing.

In this test\_collections module we are going to have several tests, many of them are for creating a collection, others are for deleting a collection, updating and so on… So *it will be better if we can organize these test cases by use case*.

For doing this we create a class called *TestCreateCollection* (*again the name of the class should start from Test otherwise it won’t work*).

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

Now every Test should have 3 parts (*which stands for AAA*),

🡪 ***Arrange*** ( *We prepare the system under test, like creating objects or put our database in initial state and so on…*)

🡪 ***Act***(*Here we kick off the behavior, we want to test*)

🡪 ***Assertion***(*Check to see if behavior we expect happens or not*)

In this particular test case, we just want to create a collection so the ***Arrange*** part is empty.

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        # Arrange

As for ***Act***in this case, we send a request to the server and for that we import *APIClient* class and create a client object.

from rest\_framework.test import APIClient

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        # Arrange

        #Act

        client = APIClient()

This class has a bunch of methods like get, post, put, delete etc. So we are going to send a post request to */store/collections/* endpoint (*first argument*) and second argument is the request body which is our collections object.

from rest\_framework.test import APIClient

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        # Arrange

        # Act

        client = APIClient()

        response = client.post("/store/collections/", {"title": "a"})

For collections object we use a dictionary here and give it a ‘title’. So we send this object to the server and got a *response* back.

Now the final part of the test is the ***Assertion***part. In this case we expect to get a 401 response from the server.

So using the *assert* statement, we can validate this.

from rest\_framework.test import APIClient

from rest\_framework import status 🡪 *import status module as well*

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        # Arrange

        # Act

        client = APIClient()

        response = client.post("/store/collections/", {"title": "a"})

        #Assertion

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

After assert we type a Boolean expression. *This response object has a property called status\_code which we check to see if this is 401 or not*.

This is how we write tests. Every test we write no matter what framework or language we use should follow the same structure (*Arrange, Act and Assertion*).

from rest\_framework.test import APIClient

from rest\_framework import status

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        client = APIClient()

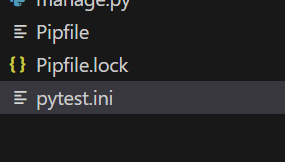
        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

**Running Tests**:

We have written our first test and now it is time to run it. But *first we need to tell pytest where our* ***settings*** *module is and for that we have to create a* ***configuration*** *file*.

In the root folder we create a new file called ***pytest.ini***.

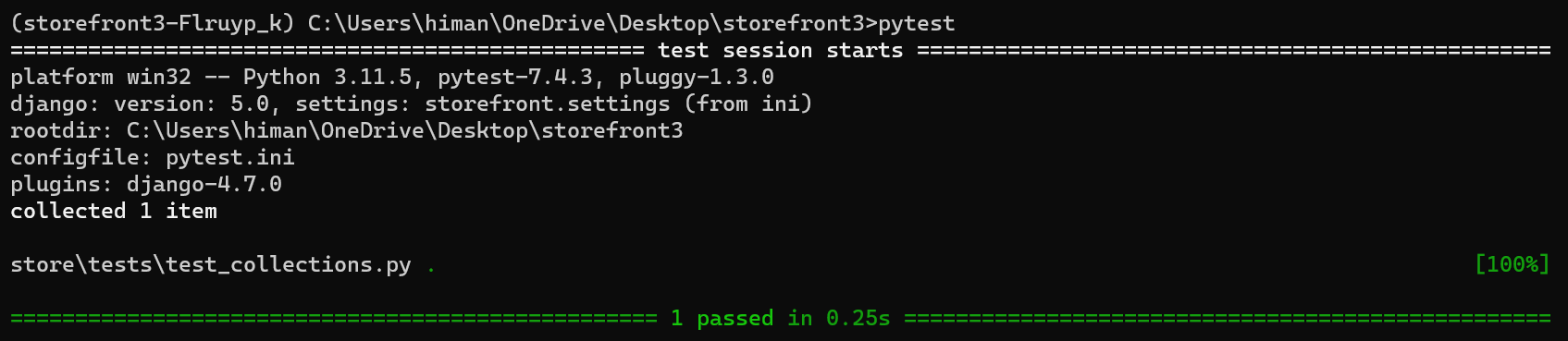


In our configuration file write,

[pytest]

DJANGO\_SETTINGS\_MODULE = storefront.settings

Now go to the terminal and write pytest:



So we can see one passing test. It is easy to write tests but we need to make sure that these tests are testing the right thing and they are telling the truth.

*How do we know this test is telling the truth*?

We are going to comment out the line of code that is responsible for making this test pass.

Here we are testing if the status code of the response is 401. So let’s go to our *CollectionViewSet*.

class CollectionViewSet(ModelViewSet):

    queryset = Collection.objects.annotate(products\_count=Count("products")).all()

    serializer\_class = CollectionSerializer

    permission\_classes = [IsAdminOrReadOnly]

    def destroy(self, request, \*args, \*\*kwargs):

        if Product.objects.filter(collection\_id=kwargs["pk"]):

            return Response(

                {

                    "error": "Collection cannot be deleted because it includes one or more products."

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

In this viewset, it is the *permission\_classes* that is helping us apply permissions. So if we comment out this line, anyone can create a collection.

If the test still passes then it means it’s a lying test. So let’s run pytest one more time.



We get failed test case, with this error, RuntimeError: Database access not allowed, use the "django\_db" mark, or the "db" or "transactional\_db" fixtures to enable it.

*Since we don’t have permission our API endpoint is trying to create a collection, so it* ***needs database access*** *and* ***by default pytest prevents that****, so we need to mark or decorate our test functions with django\_db decorator*.

So back to our test module, first we import pytest and then we decorate our function with *pytest.mark.django\_db*.

from rest\_framework.test import APIClient

from rest\_framework import status

import pytest

@pytest.mark.django\_db

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

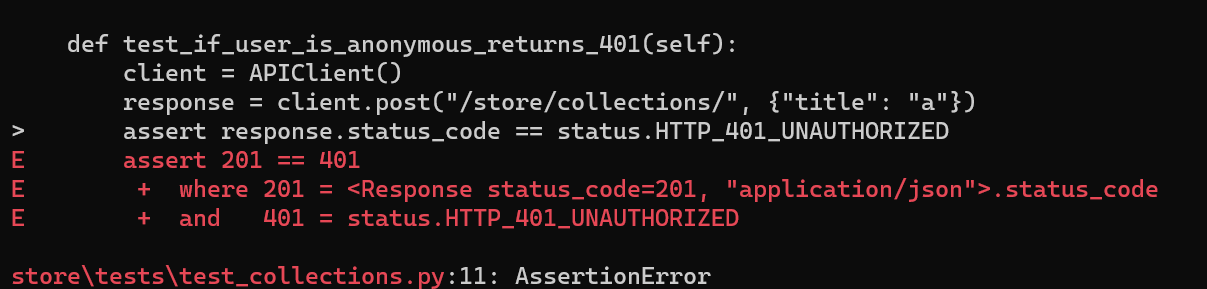
        client = APIClient()

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

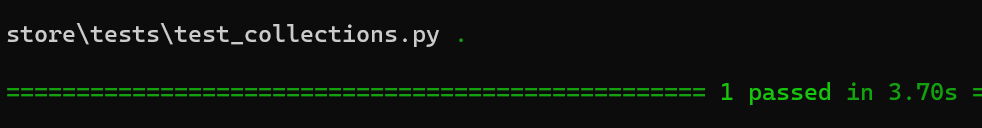
Note: we can apply this decorator to individual method but that’s repetitive so its better to apply to the class, so that all methods of this class will inherit this decorator.

After these changes let’s run pytest one more time,



We have a failure, but look at the assertion, we expected a 401 response but we got 201 (*object created*). Now we are 100% sure that this test was testing the right thing.

Back to the ViewSet, uncomment the permissions and run test again



So with this test we are testing that our endpoint is behaving correctly. Our test knows nothing about how its implemented or about our models, views, routers and so on…

Here are some ways we can execute our tests,

🡪 If we simply run pytest, it will execute all the tests in our project. So in some cases we need to isolate our tests (*only execute tests in a particular directory or in a particular module or in a particular class*)

In that case, let’s say we write the name of that particular directory like this,

pytest store/tests

It will only *execute tests in this particular directory*.

Or we can *target a particular module*:

pytest store/tests/test\_collections.py

We can also *target a specific class in this module* and for that we need to use :: (*double colons*)

pytest store/tests/test\_collections.py::TestCreateCollection

Similarly we can target a specific method in this class,

pytest store/tests/test\_collections.py::TestCreateCollection::test\_if\_user\_is\_anonymous

Now *we can also use* ***-k*** *to specify a pattern* for example, we only want to run tests that have *anonymous* in their name,

pytest -k anonymous

**Skipping tests**:

When running tests sometimes we get a failing test, but let’s say fixing that test is going to take some time and we might be in the middle of something.

In those cases *we can temporarily skip those failing tests to reduce the noise and finish our work* then when we are done we can come back and fix them.

For this demo, let’s say this is a failing test,

@pytest.mark.django\_db

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        client = APIClient()

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

We can temporarily skip it by applying a decorator @*pyest.mark.skip*.

@pytest.mark.django\_db

class TestCreateCollection:

    @pytest.mark.skip 🡪 *Here*

    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        client = APIClient()

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

Now if we run pytest on the terminal,



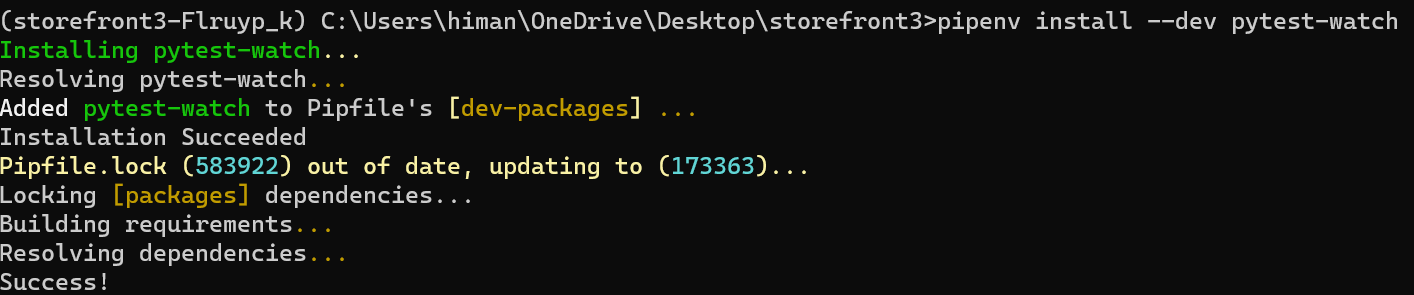
We have one skipped test.

**Continuous Testing**:

There are two ways to run tests either by on – demand (*before committing our code to git or deployment*) or we can run our tests all the time which we call *continuous testing*.

First we need to install one of the pytest plugins,

pipenv install --dev pytest-watch



With this plugin installed instead of manually running pytest every time, we can simply run ptw (*short for pytest watch*).

*This runs all our tests and every time we change our code its going to rerun the tests*.

@pytest.mark.django\_db

class TestCreateCollection:

    # @pytest.mark.skip

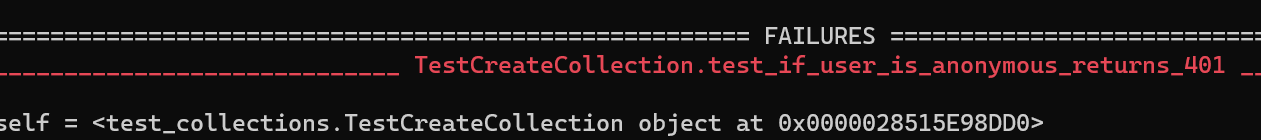
    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        client = APIClient()

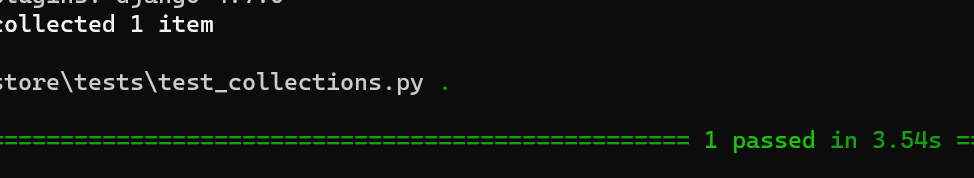
        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code != status.HTTP\_401\_UNAUTHORIZED 🡪*change*

and we see pytest starts rerunning our test and we have a failing test



After fixing it,

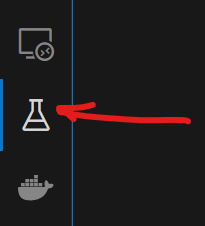


And *if we change something in our* ***ViewSet****, pytest still reruns our test*.

So as we are coding we get real time notifications about the status of our tests.

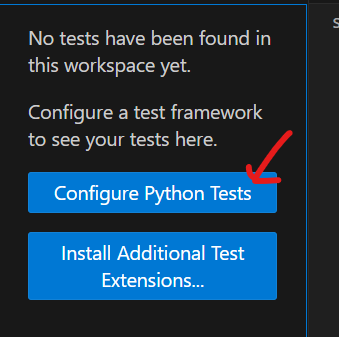
**Running and Debugging Tests in VSCode**:

We can also run and debug our tests in VSCode.



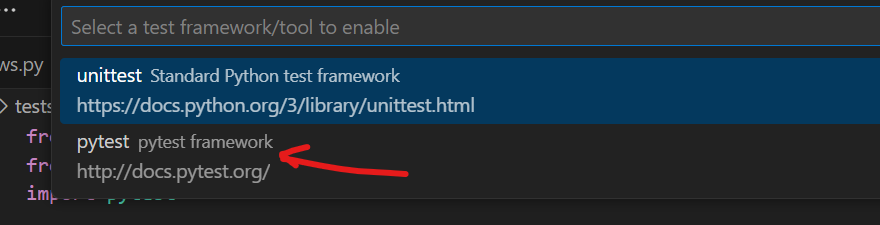
Here we have test explorer panel.

First we need to configure it to pick up our tests.

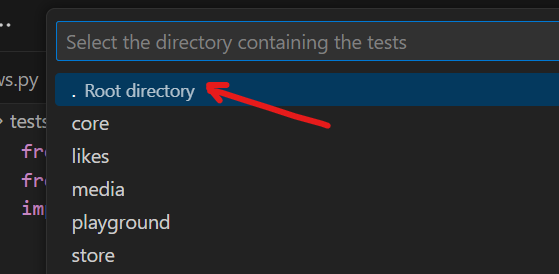


Click on configure.

Then select pytest,

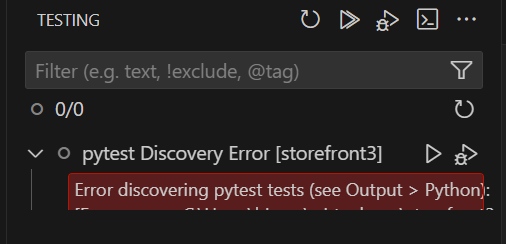


Then we select the directory that contains our test,



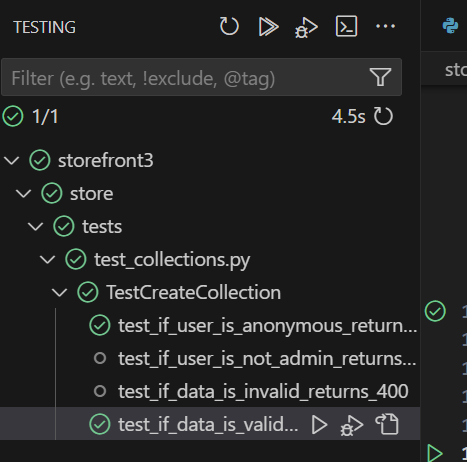
So we select root directory,

**Getting this error**:



Need to skip this lesson for now.

**Possible solution**: Run the python manage.py runserver command inside the VS code terminal, then it might work.



**Authenticating the User**:

Let us see how we can authenticate the user. So in this lesson we are going to write *a scenario where the client is authenticated but the current user is not an admin.*

First let’s add the test function name,

def test\_if\_user\_is\_not\_admin\_returns\_403(self):

We should get a 403 error if the user does not have permission to execute this operation.

Now how do we authenticate a user? The *client* object has a method called *force\_authenticate*, we call it and set user to an empty dictionary.

   def test\_if\_user\_is\_not\_admin\_returns\_403(self):

        client = APIClient()

        client.force\_authenticate(user={}) 🡪 *here*

Then we set the response and change the expected reponse status code to 403 forbidden.

    def test\_if\_user\_is\_not\_admin\_returns\_403(self):

        client = APIClient()

        client.force\_authenticate(user={})

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_403\_FORBIDDEN

Now let’s run pytest again,



And we have two passing tests.

**Single or Multiple Assertions**:

Sometimes our tests may need multiple assertions. For example for this lesson let’s write a test for the *scenario where a client is authenticated and the current user is an admin but the* ***data that we post to the server is invalid***.

def test\_if\_data\_is\_invalid\_returns\_400(self):

If data is invalid it must return 400 Bad Request error.

Here *we should authenticate the user to a user object that is an admin*. To do that first we need to import the *User* class.

from django.contrib.auth.models import User

Now we are going to *set user attribute to real User object* and we set *is\_staff* to True.

    def test\_if\_data\_is\_invalid\_returns\_400(self):

        client = APIClient()

        client.force\_authenticate(user=User(is\_staff=True))

Note: This User object is just an object in memory, it does not exist in Database. As long as we pass a User object here, *force authenticate will take care of authenticating that User*.

In this test, our focus is on invalid data, so to emphasize that, we will set the title to an empty string,

    def test\_if\_data\_is\_invalid\_returns\_400(self):

        client = APIClient()

        client.force\_authenticate(user=User(is\_staff=True))

        response = client.post("/store/collections/", {"title":""}) 🡪 *here*

Then we change the assertion to 400\_BAD\_REQUEST.

    def test\_if\_data\_is\_invalid\_returns\_400(self):

        client = APIClient()

        client.force\_authenticate(user=User(is\_staff=True))

        response = client.post("/store/collections/", {"title": ""})

        assert response.status\_code == status.HTTP\_400\_BAD\_REQUEST

This is our first assertion but we need another assertion for the error message that we expect in the body of the response.

So,

    def test\_if\_data\_is\_invalid\_returns\_400(self):

        client = APIClient()

        client.force\_authenticate(user=User(is\_staff=True))

        response = client.post("/store/collections/", {"title": ""})

        assert response.status\_code == status.HTTP\_400\_BAD\_REQUEST

        assert response.data['title'] is not None *🡪 Second assertion*

response.data is a dictionary, so we can access the title property using square bracket notation and check if this is not None.

Note: We don’t care about the actual error message because that’s implementation detail and wordings might change in the future. We don’t want our tests to break if we change an error message. As long as we have something for title in the response, our tests will pass.

In this example, we have multiple assertions. But both these assertions are logically related (*Testing the status code and body of the response*). So its not breaking the fundamentals of testing which is…



So Tests should have single responsibility but *they may have multiple assertions as long as they are logically related*.

Let’s write another test for the *scenario where the data is valid*.

So we have a valid user here who is sending valid data here, so the status code should be 201\_CREATED and in the body of the response we should have the id of the new collection.

    def test\_if\_data\_is\_valid\_return\_201(self):

        client = APIClient()

        client.force\_authenticate(user=User(is\_staff=True))

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_201\_CREATED

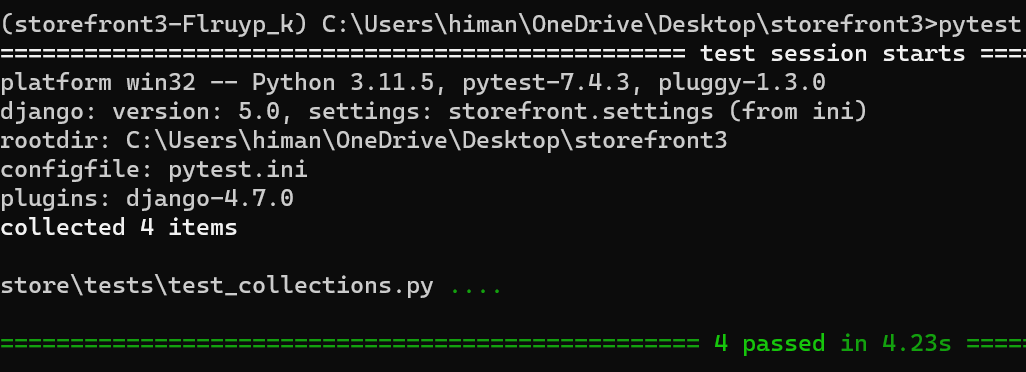
        assert response.data['id'] > 0 🡪 *Here*

Note: some people take a different approach. Instead of checking that the id is greater than 0, they go into the database and pull the collection with that id to ensure that collection was saved.

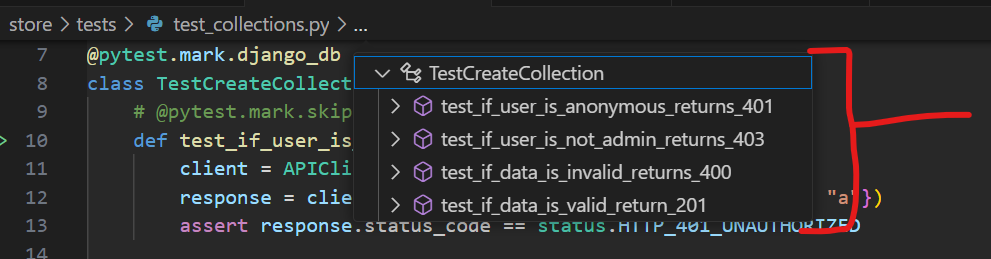
In theory it may sound like reasonable approach but no so much in practice. Because if we use the Collection model to fetch that collection our tests becomes coupled with our implementation.

So tomorrow if we change that model class, the tests that are coupled with them might be affected. *The less our tests know about the internals of our system the more reliable they are going to be*.

Now, run all these tests,



Note: In VS code we can see all the test cases for creating a collection.

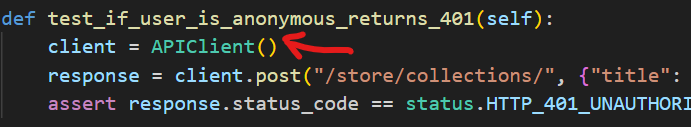


It is useful, we can easily navigate between our tests.

**Fixtures**:

Let’s talk about a powerful feature in pytest called ***fixtures***. Using fixtures *we can remove duplication in our test code*.

Notice in every test case we have written so far,



first we have to create a client object and that means in every test module first we have to import *APIClient* class so we can create a client object in each test which is repetitive. This is where we can use *fixtures*.

So here in the tests folder, we will create a new file called *conftest*.py. This is a special file for pytest. So *the fixtures or reusable functions that we define here, pytest will automatically load them without us having to explicitly import this module*.

In our conftest.py file,

from rest\_framework.test import APIClient

import pytest

def api\_client():

    return APIClient()

In *api\_client*() function we just defined, we are returning an instance of APIClient class.

To make this function a fixture, we apply a fixture decorator to it,

from rest\_framework.test import APIClient

import pytest

@pytest.fixture 🡪 *Here*

def api\_client():

    return APIClient()

Now this function is a reusable piece of code and we can add it to each test as a parameter.

In our test\_collections.py file,

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client): 🡪 *here*

        response = client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

We removed our client = APIClient() line and add *api\_client* as a parameter to this function.

So *when pytest tries to execute this test, it will look at our fixtures that we have defined in conftest.py and figure out that we have a function with this name and returns its value here*.

Instead of client we have access to *api\_client*…

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client):

        response = api\_client.post("/store/collections/", {"title": "a"}) 🡪*here*

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

As a result we removed one unnecessary line from this test. We can do the same with other tests.

@pytest.mark.django\_db

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client):

        response = api\_client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

    def test\_if\_user\_is\_not\_admin\_returns\_403(self, api\_client):

        api\_client.force\_authenticate(user={})

        response = api\_client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_403\_FORBIDDEN

    def test\_if\_data\_is\_invalid\_returns\_400(self, api\_client):

        api\_client.force\_authenticate(user=User(is\_staff=True))

        response = api\_client.post("/store/collections/", {"title": ""})

        assert response.status\_code == status.HTTP\_400\_BAD\_REQUEST

        assert response.data["title"] is not None

    def test\_if\_data\_is\_valid\_return\_201(self, api\_client):

        api\_client.force\_authenticate(user=User(is\_staff=True))

        response = api\_client.post("/store/collections/", {"title": "a"})

        assert response.status\_code == status.HTTP\_201\_CREATED

        assert response.data["id"] > 0

Now every test is getting api\_client as a parameter.

Second repetition is in calling this endpoint (*/store/collections/*). The only difference is in object that we pass to the server (*sometimes valid and sometimes invalid collection data*). Let’s see how we can use a fixture to simplify this code.

We will not define this fixture in conftest.py file because here we define fixtures that we are going to reuse across test modules (*since creating a collection is very specific to test\_collections module*).

So we will define this fixture in our test\_collections.py itself,

def create\_collection(api\_client):

Here we define a function called *create\_collection*(), since this function need api\_client (*to post the request to the server*) so we can add it as a parameter and again pytest will automatically pass that fixture to this function.

Since this create\_collection function itself is going to be a fixture so we can @*pytest.fixture* decorator.

@pytest.fixture

def create\_collection(api\_client):

Here we should call api\_client with post to our collections endpoint. But when passing the request object we don’t want to hardcode any collection object here (*sometimes valid and sometimes invalid request*).

**Wrong way**:

If we pass collection as a parameter to create\_collection fixture,

@pytest.fixture

def create\_collection(api\_client, collection):

    api\_client.post('/store/collections/', collection)

Pytest will think ‘*collection*’ is also a fixture just like *api\_client* and try to find it.

**Right way**:

Create an inner function which is going to take this *collection* as a parameter and return our inner function from outer function.

@pytest.fixture

def create\_collection(api\_client): 🡪 *Outer function our fixture*

    def do\_create\_collection(collection): 🡪 *Inner function*

      return api\_client.post("/store/collections/", collection)

    return do\_create\_collection

Understanding this code:

🡪 We created a fixture called *create\_collection*. In this function instead of returning an object we are returning a function (*do\_create\_collection*).

Now if we go to one of our test and add our *create\_collection* fixture.

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client, create\_collection): 🡪 *Here*

Before pytest run this test, its going to call this function / fixture which we know returns another function.

So we give this function our object,

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client, create\_collection):

        create\_collection( {"title": "a"}) 🡪 *Here*

And save the response,

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, api\_client, create\_collection):

        response = create\_collection({"title": "a"}) 🡪 *Here*

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

Let’s make these changes in all the test cases,

@pytest.mark.django\_db

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, create\_collection):

🡪 *don’t need api\_client here because no need to authenticate*

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

    def test\_if\_user\_is\_not\_admin\_returns\_403(self, api\_client, create\_collection):

        api\_client.force\_authenticate(user={})

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_403\_FORBIDDEN

    def test\_if\_data\_is\_invalid\_returns\_400(self, api\_client, create\_collection):

        api\_client.force\_authenticate(user=User(is\_staff=True))

        response = create\_collection({"title": ""})

        assert response.status\_code == status.HTTP\_400\_BAD\_REQUEST

        assert response.data["title"] is not None

    def test\_if\_data\_is\_valid\_return\_201(self, api\_client, create\_collection):

        api\_client.force\_authenticate(user=User(is\_staff=True))

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_201\_CREATED

        assert response.data["id"] > 0

*Exercise*:

Define a global fixture for authenticating users in the conftest.py module.

@pytest.fixture

def authenticate(api\_client):

    def do\_authenticate(is\_staff=False):

        return api\_client.force\_authenticate(user=User(is\_staff=is\_staff))

    return do\_authenticate

In our conftest.py module we have created a fixture called *authenticate* which takes *api\_client* as a parameter.

In this fixture we return an inner function *do\_authenticate* which takes a keyword argument called is\_staff which is set to False by default (*so we don’t have to specify that value in every test*).

In inner function we call *force\_authenticate* which takes a ***User*** object (*so we don’t have to import User class in every test module*).

This User object has is\_staff argument which we set to is\_staff value we receive in the inner function.

Back to our test module,

    def test\_if\_user\_is\_not\_admin\_returns\_403(self, create\_collection, authenticate):

        authenticate() 🡪 *is\_staff is false by default, can simply call function*

response = create\_collection({"title": "a"})

assert response.status\_code == status.HTTP\_403\_FORBIDDEN

Note: How we have formatted this test case now,

🡪 ***Arrange***

    def test\_if\_user\_is\_not\_admin\_returns\_403(self, create\_collection, authenticate):

        authenticate()

Preparing our system.

🡪 ***Act***

response = create\_collection({"title": "a"})

Preparing our response

🡪 ***Assert***

assert response.status\_code == status.HTTP\_403\_FORBIDDEN

Preparing our assertion.

All test cases now,

@pytest.mark.django\_db

class TestCreateCollection:

    def test\_if\_user\_is\_anonymous\_returns\_401(self, create\_collection):

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_401\_UNAUTHORIZED

    def test\_if\_user\_is\_not\_admin\_returns\_403(self, authenticate, create\_collection):

        authenticate()

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_403\_FORBIDDEN

    def test\_if\_data\_is\_invalid\_returns\_400(self, authenticate, create\_collection):

        authenticate(is\_staff=True)

        response = create\_collection({"title": ""})

        assert response.status\_code == status.HTTP\_400\_BAD\_REQUEST

        assert response.data["title"] is not None

    def test\_if\_data\_is\_valid\_return\_201(self, authenticate, create\_collection):

        authenticate(is\_staff=True)

        response = create\_collection({"title": "a"})

        assert response.status\_code == status.HTTP\_201\_CREATED

        assert response.data["id"] > 0

**Creating Model Instances**:

So are done for creating tests for creating a collection, now let’s write a test for retrieving a collection.

We will begin by creating a class for organizing the tests for retrieving a collection, let’s call it *TestRetrieveCollection*.

@pytest.mark.django\_db

class TestRetrieveCollection:

Now there are two scenarios we need to test here, one is that **collection does not exist** (*we expect 404 error*) and the other scenario is that the **collection exists** (*we expect 200 response with collection object in the body of the response*).

Let’s create a method called, *test\_if\_collection\_exists\_returns\_200*(), here *we need api\_client to send a get request to the server*. This is the part which is different from previous tests.

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

In the *arrange* part of the test, *we need to create a collection so we can retrieve it later*, because this test should not be dependent on other tests that create a collection (*because if this execute first, it’s going to fail since there is no collection in the database*).

So our tests should not have any dependency on each other, **we should treat each test as if it’s the only test in the world**.

From first glance, we have to choices on how to create a collection:

1. Choice #1:

Use api\_client to send a post request to ‘/store/collections/’ endpoint.

   def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        api\_client.post('/store/collections/') 🡪 *Here*

But the problem with this approach is that if there is a bug while creating a collection, this line is going to fail and eventually our test will fail (*even though retrieving a collection actually works*).

1. Choice #2:

We can use the *Collection* model with *create* method.

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        Collection.objects.create(title = 'a')

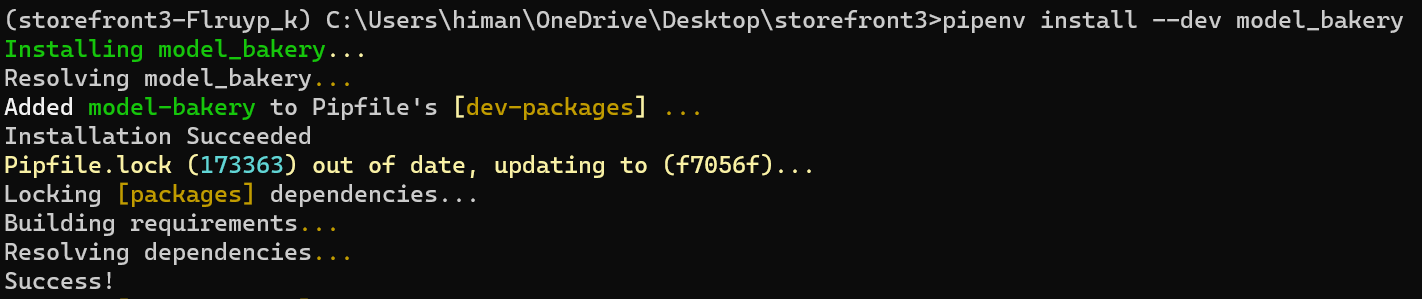
Note: Earlier we decided on the principle that when writing tests, we should test the behavior and not the implementation. This *Collection model is part of the implementation, not part of the interface of our API*. So in this case *we are violating that principle*.

*The other issue is defining fields*, we know that every model whether Collection or Product has a lot of fields and if we would define all those fields here while creating a collection or product that would create a bit of noise in this test.

So *there is a better way*:

Here we use a fantastic library called ***model\_bakery***.

pipenv install --dev model\_bakery



Now we have model\_bakery installed, so let’s import *baker* module in our test\_collections.py.

from model\_bakery import baker

With this approach, *we no longer need to initialize individual properties of our model*, it will taken care by baker module.

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection) 🡪 *here*

So for each field depending on its type, it will give a random value.

To see these random values, let’s print *collection* on the terminal where we access it’s *\_\_dict\_\_* property.

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        print(collection.\_\_dict\_\_) 🡪 *Here*

Note: When running tests, these print statements don’t work unless test fails. So to see this print statement we are going to deliberately fail this test.

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        print(collection.\_\_dict\_\_)

        assert False 🡪 *set assert to False*

We see a collection object with a random id, title.



Notice title is a long random string.

*If we had other fields here like DateTime, Integer, Text etc.…model\_bakery would automatically initialize them to some value*.

Note: model\_bakery also takes care of relationship for us, so for example, if we say *baker.make*(*Product*) it will automatically create a product and a collection because each product should be inserted into one and only one collection.

In baker.make method we have a special keyword argument called *\_quantity* which can create as many objects in the database.

baker.make(Product, \_quantity = 10)

🡪 *10 products will be created in database*

But there is a problem, each product will have separate *collection\_id*. What if we want to keep all these products in the same collection.

For doing this we are going to explicitly set some of the values here.

 collection = baker.make(Collection) 🡪 *First create a collection*

baker.make(Product, collection=collection, \_quantity = 10)

🡪 *set the collection value to our collection*

This is the basics of model\_bakery, if need to learn more about it <https://model-bakery.readthedocs.io/en/latest/>

Back to our test, in the *act* part we send a get request to ‘/store/collections/’

@pytest.mark.django\_db

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        response = api\_client.get(f"/store/collections/{collection.id}") 🡪*here*

We need to add collection\_id here, so we used formatted string.

Next we create two *assertions*, first the response code

@pytest.mark.django\_db

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        response = api\_client.get(f"/store/collections/{collection.id}")

        assert response.status\_code == status.HTTP\_200\_OK

And then collection object which is in the body of the response. Since *response.data* is a dictionary, we can access the individual properties like *response.data[‘id’] = collection.id*.

But there is a better way, we can compare the entire data object with another dictionary.

@pytest.mark.django\_db

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        response = api\_client.get(f"/store/collections/{collection.id}")

        assert response.status\_code == status.HTTP\_200\_OK

        assert response.data == { 🡪 *here*

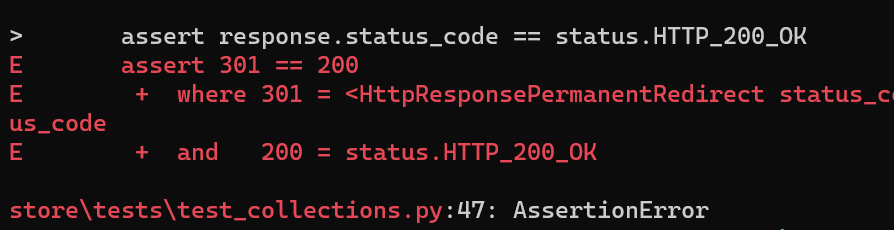
            "id": collection.id,

            "title": collection.title,

        }

Let’s run pytest,

We have a failure because the status of the response is 301 instead of 200.



301 HTTP response code means a *permanent redirect*. In Django applications URL must terminate with a forward slash / . So …

@pytest.mark.django\_db

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        response = api\_client.get(f"/store/collections/{collection.id}/")

        assert response.status\_code == status.HTTP\_200\_OK

        assert response.data == {

            "id": collection.id,

            "title": collection.title,

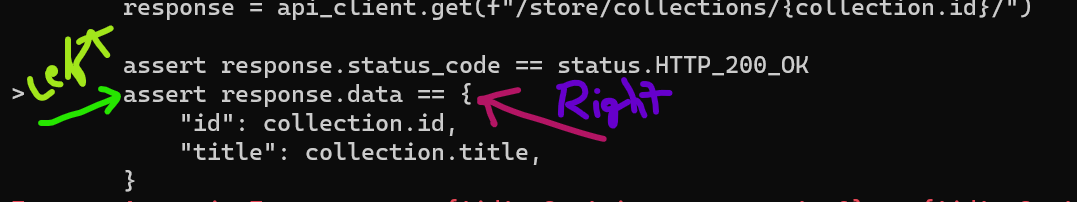
        }

Let’s run our test one more time,

We get another error, saying Left contains one more item.



This is left and right object,



So let’s add *products\_count* field to the right,

@pytest.mark.django\_db

class TestRetrieveCollection:

    def test\_if\_collection\_exists\_returns\_200(self, api\_client):

        collection = baker.make(Collection)

        response = api\_client.get(f"/store/collections/{collection.id}/")

        assert response.status\_code == status.HTTP\_200\_OK

        assert response.data == {

            "id": collection.id,

            "title": collection.title,

            "products\_count": 0,

        }

Now our test passes,



Note: When we run our tests, pytest will automatically create a test database for us.

DATABASES = {

    "default": {

        "ENGINE": "django.db.backends.mysql",

        "NAME": "storefront3",

        "HOST": "localhost",

        "USER": "root",

        "PASSWORD": "S@b8at1ze",

    }

}

Look here we have a database called storefront3. When we run our tests, pytest will automatically create a database called test\_storefront3.

*It will create this database at the beginning of executing all these tests and when test execution is finished, pytest will drop this database*.

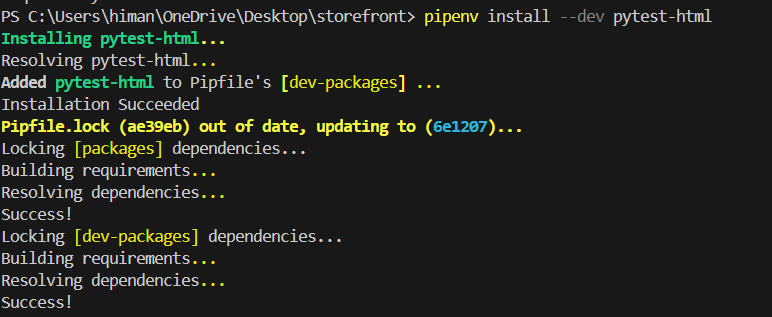
So this way the data that we create while testing is not going to get mixed up with the data that we have in our development database.

**Exercise**:

Write all tests for collections endpoint as well as products endpoint.

**A normal Report generation**:

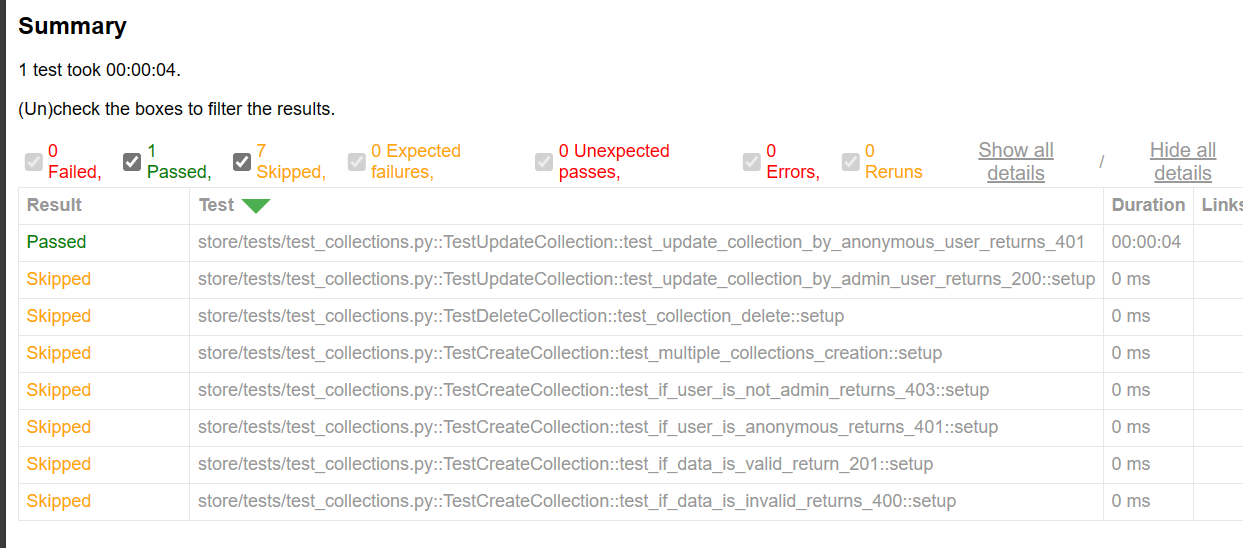
For generating HTML reports for these tests install *pytest-html* package  
pipenv install --dev pytest-html



Then,  
pytest --html=report.html

For generating a *report.html* file in the directory.

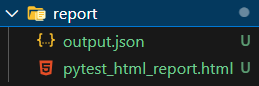
This file gives a report like this…



**A better report generation**:

As you can see this is a very bland report, we can use another plugin as well called *pytest-html-reporter*.  
pipenv install --dev pytest-html-reporter

Then you create a folder called *reports* and give this a custom file name  
pytest --html-report=./report --title='My Custom Report Title'



It will create a file like this…

This report has everything from dashboard to test metrics view and etc.

