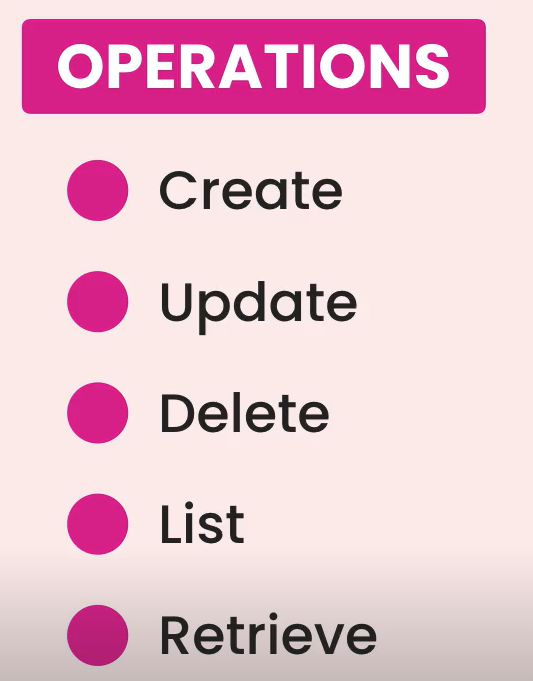




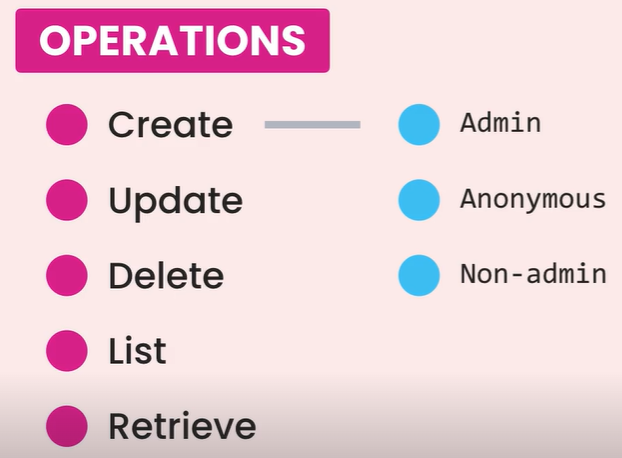
**What is Automated testing** :

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.



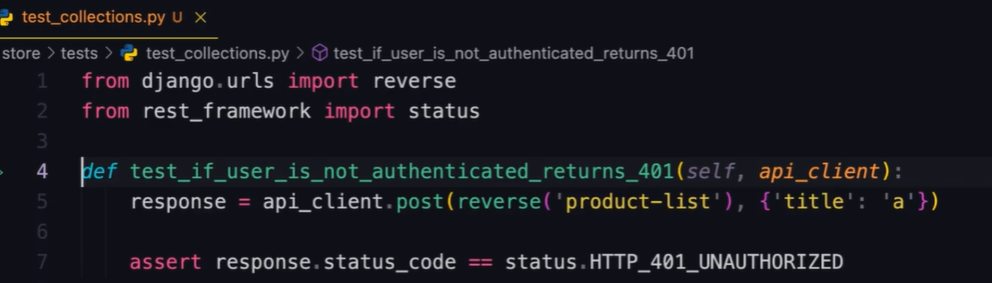
Plus over time we are going to forget various rules hidden behind these endpoints, unless we document them in a scripts that a tester can use to test various scenarios.

This is where *automated testing* comes to rescue,

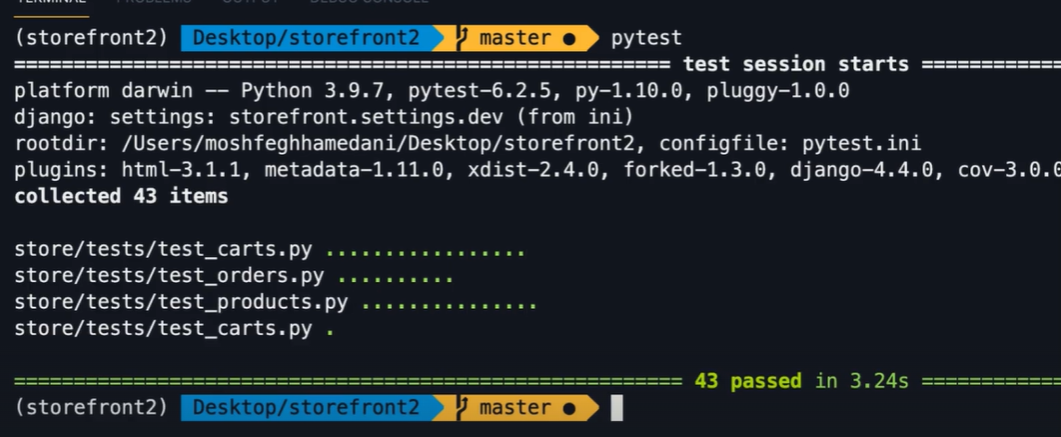


With automated testing we can write code to test our endpoints and their business rules.

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.

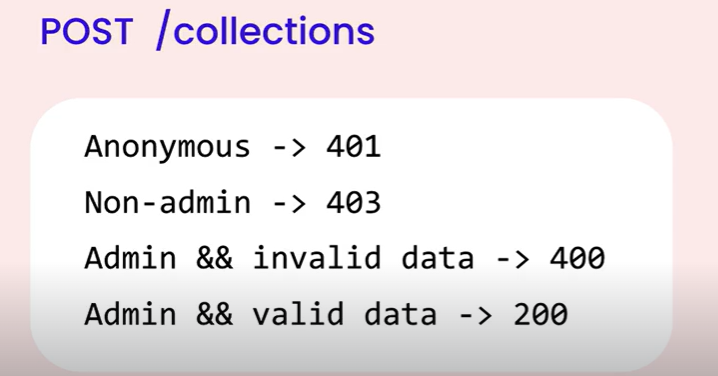
These are implementation details and they are prone to change.

*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.

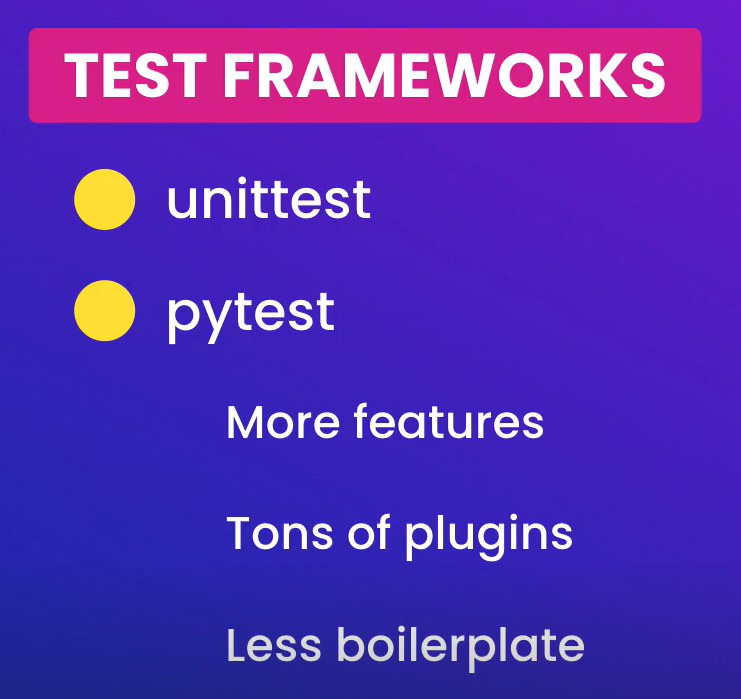
So let’s emphasize it one more time,



**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.



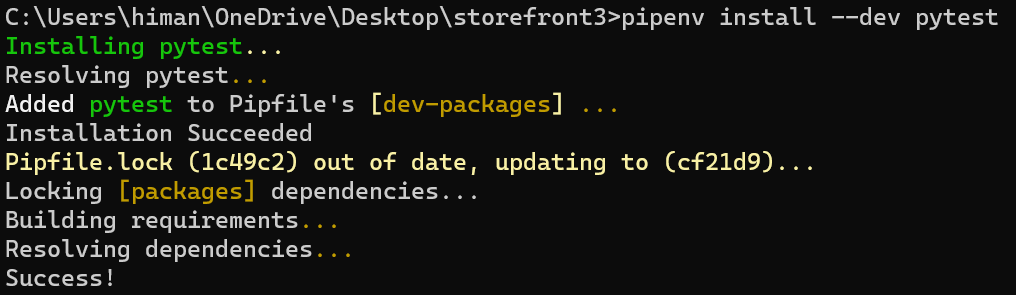
Here is side by side comparison of the same test written with *unittest* and *pytest*,



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

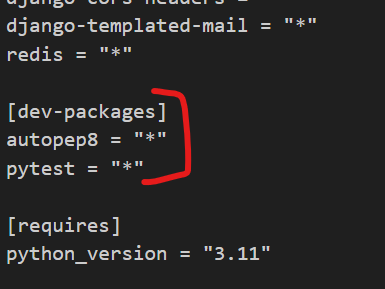
So in the terminal we are going to install *pytest*,

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*).

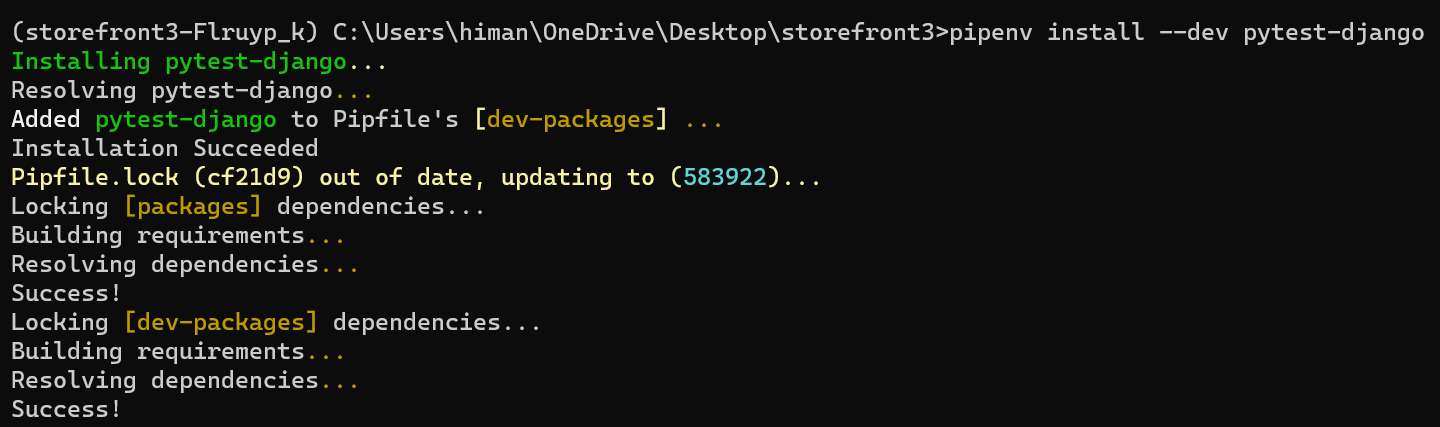
If we go to our *pipfile*,



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 = "\*"

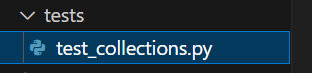
pytest = "\*"

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()

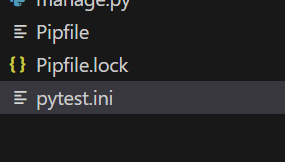
        response = client.post("/store/collections/", {"title": "a"})

        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***.

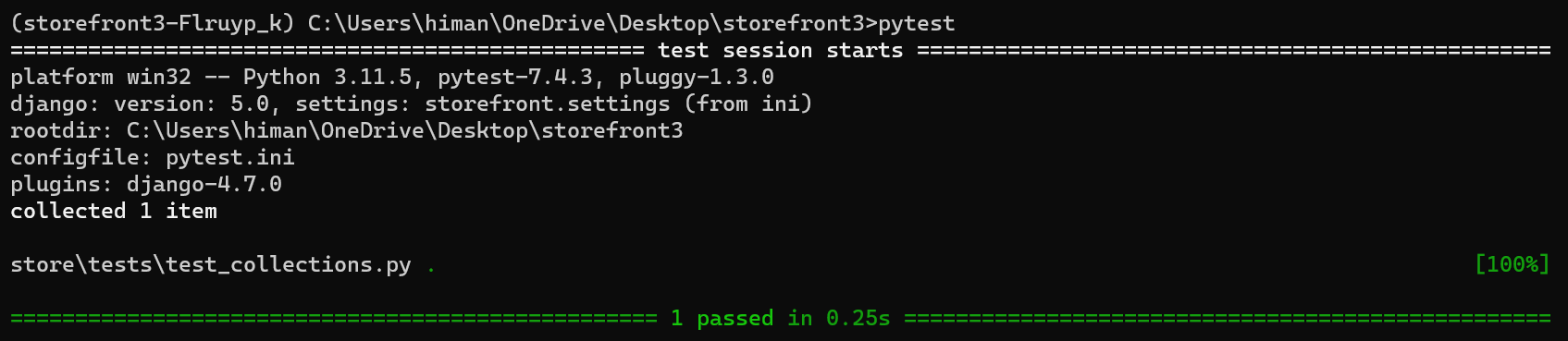


In our configuration file write,

[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."

                },

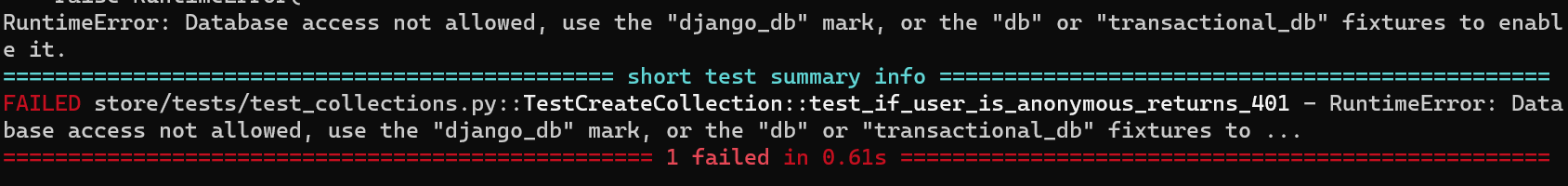
                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        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):

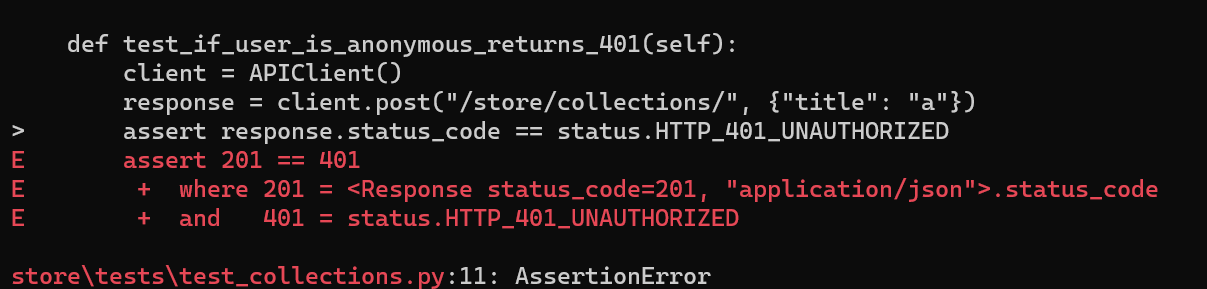
        client = APIClient()

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

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

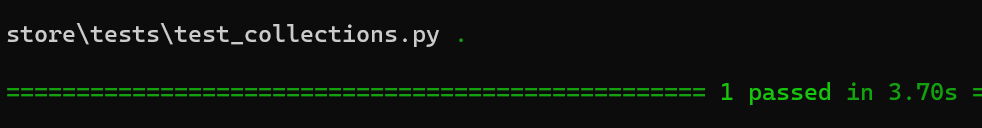
Note: we can apply this decorator to individual method but that’s repetitive so its better to apply to the class, so that all methods of this class will inherit this decorator.

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*

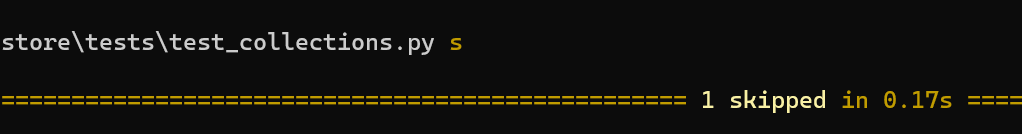
    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        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,



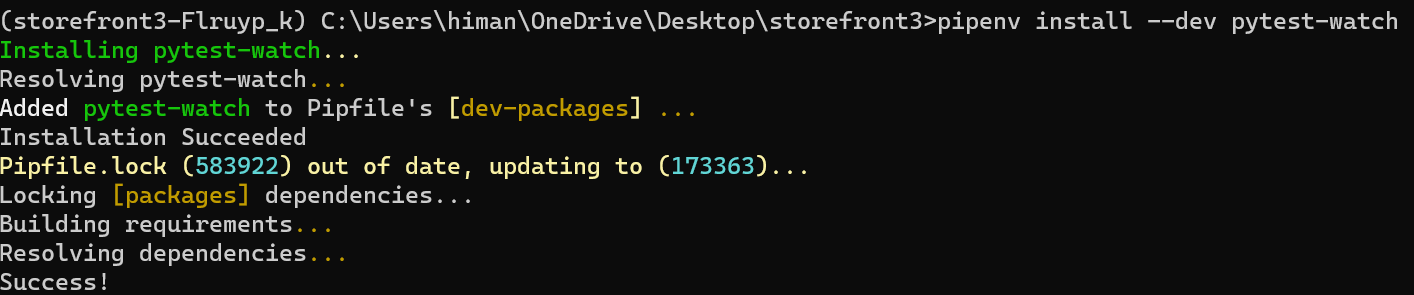
We have one skipped test.

**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

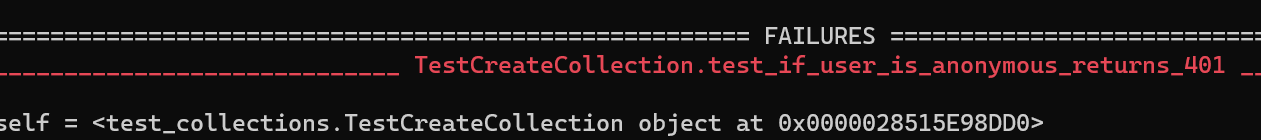
    def test\_if\_user\_is\_anonymous\_returns\_401(self):

        client = APIClient()

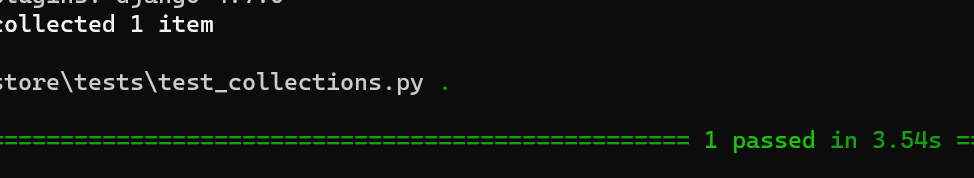
        response = client.post("/store/collections/", {"title": "a"})

        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,

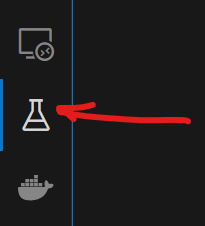


And *if we change something in our* ***ViewSet****, pytest still reruns our test*.

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

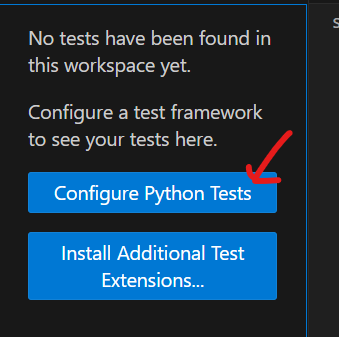
**Running and Debugging Tests in VSCode**:

We can also run and debug our tests in VSCode.



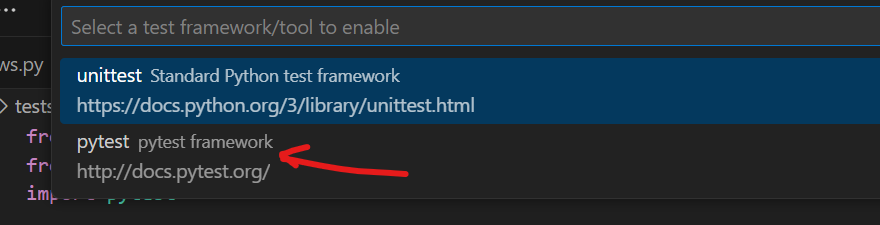
Here we have test explorer panel.

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

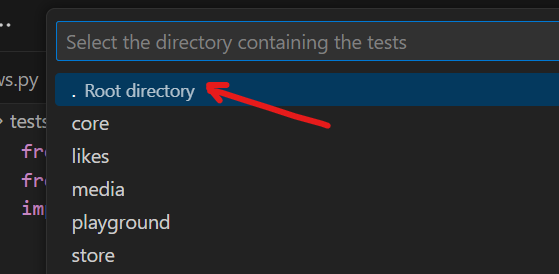


Click on configure.

Then select pytest,



Then we select the directory that contains our test,



So we select root directory,