# Complaint’s system – testing

## Introduction

Today will aim to ensure our project is working correctly and it is tested against critical business criteria such as submitting a complaint, approving a complaint, and logging in.

Sadly, in the real world, especially in startup companies, testing is highly underestimated. Not because the developers and managers do not know the value of having tests, but because there is not enough money and time. They try to go on the market with a ready product as soon as possible, and tests take 1/3 of the time required for the development.

Having a product without tests is not a question IF a serious bug would be missed but WHEN it will be discovered. If you can, always add tests.

We will start with some preparation, and then we will test from request to response with different scenarios.

## Set up tests

pip install flask\_testing pytest

We will use this library and pytest for our testing suits.

Before we start, we need to abstract our existing code a little bit, so that we can test our application properly.

In config.py

…

class TestingConfig:  
 *"""Configurations for Testing, with a separate test database."""* TESTING = True  
 SQLALCHEMY\_DATABASE\_URI = **f"postgresql:// {put your db user}:{put your db password} @localhost: {put your db port} /{put your db name} "** DEBUG = True  
  
  
def create\_app(config=**'config.DevelopmentConfig'**):  
 app = Flask(\_\_name\_\_)  
 app.config.from\_object(config)  
  
 api = Api(app)  
 migrate = Migrate(app, db)  
 CORS(app)  
 [api.add\_resource(\*route) for route in routes]  
 return app

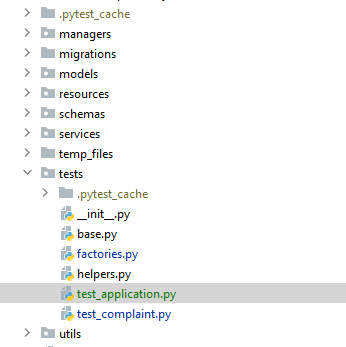
We will create new configuration class where we will connect a new db which is created for testing (you should never use production or staging database for testing).

We moved the app creation in create\_app function. Now we will adjust the main.py file:

from config import create\_app  
from db import db  
  
app = create\_app()  
  
  
@app.before\_first\_request  
def create\_tables():  
 db.init\_app(app)  
 db.create\_all()  
  
  
@app.after\_request  
def close\_request(response):  
 db.session.commit()  
 return response  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run()

Create a python package in the root of the project called **tests**. It is important, because pytest has a naming convention – the folder must be tests and each test file must start with test\_{name}.py, each test class should start with Test{NameOfTheClass} and each method should start with test\_{name\_of\_the\_method}. This is important, otherwise it won’t be recognized as a test from pytest runner.

We will create a couple of other files as well: **base.py, factories.py, helpers.py, test\_application.py, test\_complaint.py**.



We will start by adding some logic in base that we know might come handy.

This will serve us well when we need to mock uuids or authenticate users when we are testing the complaint.

from managers.auth import AuthManager  
  
  
def generate\_token(user):  
 return AuthManager.encode\_token(user)  
  
  
def mock\_uuid():  
 return **"11111111-1111-1111-1111-111111111111"**

We will use factories to make our required data for the tests easier to be created – if we test a complaint, we have to have a complainer who is submitting the complaint and he/she should have a token.

pip install factory\_boy

This library will help us create random data attributes for our models. Docs are [here](https://factoryboy.readthedocs.io/en/stable/).

from db import db  
from random import randint  
  
import factory  
  
from models import ComplainerModel, RoleType  
  
  
class BaseFactory(factory.Factory):  
 @classmethod  
 def create(cls, \*\*kwargs):  
 object = super().create(\*\*kwargs)  
 db.session.add(object)  
 db.session.flush()  
 return object  
  
  
class ComplainerFactory(BaseFactory):  
 class Meta:  
 model = ComplainerModel  
  
 id = factory.Sequence(lambda n: n)  
 first\_name = factory.Faker(**"first\_name"**)  
 last\_name = factory.Faker(**"last\_name"**)  
 email = factory.Faker(**"email"**)  
 phone = str(randint(100000, 200000))  
 password = factory.Faker(**"password"**)  
 role = RoleType.complainer  
 iban = factory.Faker(**"iban"**)

Here we inherit Factory in BaseFactory class, only to extend the create method of this class. Our goal would be to ensure that after the object is created we will add it to the testing database and we will return the object.

The ComplainerFactory inherits the BaseFactory which means each time a ComplainerFactory object is created it will go to its base create and save the object to the database.

With the meta class we specify the model against these attributes and their values should be created. In our case this is the ComplainerModel.

The attributes such as names, email and iban could be created with Faker – this will give us a combination of valid names and attributes. It is very convenient, and free up a time for actual testing and we won’t spend time thinking of values on each test.

In helpers.py file we put everything that might come useful – for example we will store a variable with already encoded string and we will use it when we send a complaint’s data for creation.

encoded\_file = {put your encoded string here}

## Application tests

In test\_application. py we will make sure we are testing the overall application behavior such as returning 401 for endpoints that are protected:

import json  
  
from flask\_testing import TestCase  
  
from config import create\_app  
from db import db  
  
  
class TestApp(TestCase):  
 def create\_app(self):  
 return create\_app(**"config.TestingConfig"**)  
  
 def setUp(self):  
 db.init\_app(self.app)  
 db.create\_all()  
  
 def tearDown(self):  
 db.session.remove()  
 db.drop\_all()  
  
 def test\_protected(self):  
 for method, url in [  
 (**"PUT"**, **"/approvers/complaints/1/approve"**),  
 (**"PUT"**, **"/approvers/complaints/1/reject"**),  
 (**"GET"**, **"/complainers/complaints"**),  
 (**"POST"**, **"/complainers/complaints"**),  
 (**"POST"**, **"/admins/create-admin"**),  
 (**"POST"**, **"/admins/create-approver"**),  
 (**"DELETE"**, **"/admins/complains/1"**),  
 ]:  
 if method == **"POST"**:  
 resp = self.client.post(url, data=json.dumps({}),)  
 elif method == **"GET"**:  
 resp = self.client.get(url)  
 elif method == **"PUT"**:  
 resp = self.client.put(url, data=json.dumps({}),)  
 else:  
 resp = self.client.delete(url)  
 self.assert401(resp, {**'message'**: **'Invalid or missing token'**})

We create a class TestApp and inherit from TestCase from **flask\_testing not form unitests**. You might find setup and teardown interesting. The setup is invoked before **each** **testing** **method** it ensures that for each test, we create a database and all tables. Teardown takes care to remove everything from the session and database. It is invoked **after** each test. They are **must** for each testing class. The create app is interesting. We create a flask app with testing configurations we did earlier.

We use the self.cleint to make a request. When we do post or put always the data should be convert to json using

json.dumps({the dict data we want to convert})

There are different assertions for status code and messages provided from TestCase. In this case we use 401, because we make a request without being authenticated and we expect a 401 Unauthorized status code would be returned.

Your **home assignment** would be to write testing methods for different cases – such as: user is authenticated but with invalid token, user is authenticated with valid token but does not have permission to access this endpoint.

For example – let’s test the URLs for delete complaint and create admin/approver which should be accessed only by admins:

We will create another method test\_protected\_admin\_endpoints\_require\_admin\_rights and we will create a complainer using the factory we did earlier. We will generate a token against complainer (not admin) and we will try to access admin endpoints as complainers. The error would be different – 403 forbidden – which means we are authorized correctly but we just do not have the required role (authorization).

Think of all possible cases that you can and add testing methods for them in this class.

import json  
  
from flask\_testing import TestCase  
  
from config import create\_app  
from db import db  
from tests.base import generate\_token  
from tests.factories import ComplainerFactory  
  
  
class TestApp(TestCase):  
 def create\_app(self):  
 return create\_app(**"config.TestingConfig"**)  
  
 def setUp(self):  
 db.init\_app(self.app)  
 db.create\_all()  
  
 def tearDown(self):  
 db.session.remove()  
 db.drop\_all()  
  
 def test\_protected(self):  
 for method, url in [  
 (**"PUT"**, **"/approvers/complaints/1/approve"**),  
 (**"PUT"**, **"/approvers/complaints/1/reject"**),  
 (**"GET"**, **"/complainers/complaints"**),  
 (**"POST"**, **"/complainers/complaints"**),  
 (**"POST"**, **"/admins/create-admin"**),  
 (**"POST"**, **"/admins/create-approver"**),  
 (**"DELETE"**, **"/admins/complains/1"**),  
 ]:  
 if method == **"POST"**:  
 resp = self.client.post(url, data=json.dumps({}),)  
 elif method == **"GET"**:  
 resp = self.client.get(url)  
 elif method == **"PUT"**:  
 resp = self.client.put(url, data=json.dumps({}),)  
 else:  
 resp = self.client.delete(url)  
 self.assert401(resp, {**'message'**: **'Invalid or missing token'**})  
  
 def test\_protected\_admin\_endpoints\_require\_admin\_rights(self):  
 for method, url in [  
 (**"POST"**, **"/admins/create-admin"**),  
 (**"POST"**, **"/admins/create-approver"**),  
 (**"DELETE"**, **"/admins/complains/1"**),  
 ]:  
 complainer = ComplainerFactory()  
 token = generate\_token(complainer)  
 headers = {**"Authorization"**: **f"Bearer** {token}**"**}  
 if method == **"POST"**:  
 resp = self.client.post(url, data=json.dumps({}),headers=headers)  
 elif method == **"DELETE"**:  
 resp = self.client.delete(url, headers=headers)  
 expected\_message = {**'message'**: **'You do not have the rights to access this resource'**}  
 self.assert\_403(resp, expected\_message)

## Complaints tests – mocking

from flask\_testing import TestCase  
  
from db import db  
from config import create\_app

class TestComplaint(TestCase):  
 url = **"/complainers/complaints"** def create\_app(self):  
 return create\_app(**"config.TestingConfig"**)  
  
 def setUp(self):  
 db.init\_app(self.app)  
 db.create\_all()  
  
 def tearDown(self):  
 db.session.remove()  
 db.drop\_all()

In test\_complaint.py we will start by create a class TestComplaint, we will add the setup, theardown and create app method.

Now we want to test our schemas – we want to ensure that if we miss a required field the schema would raise an exception:

…

def test\_complaint\_missing\_input\_fields\_raises(self):  
 comp = ComplainerFactory()  
 token = generate\_token(comp)  
  
 complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 0)  
 data = {  
 **"title"**: **"Test"**,  
 **"description"**: **"Test test"**,  
 **"photo"**: encoded\_file,  
 **"photo\_extension"**: **"png"**,  
 **"amount"**: 10.00,  
 }  
  
 for key in data:  
 current\_data = data.copy()  
 current\_data.pop(key)  
 resp = self.client.post(  
 self.url,  
 data=json.dumps(current\_data),  
 headers={  
 **"Content-Type"**: **"application/json"**,  
 **"Authorization"**: **f"Bearer** {token}**"**,  
 },  
 )  
  
 message = resp.json[**"message"**]  
 expected\_message = **"Invalid fields {'"** + key + **"': ['Missing data for required field.']}"** self.assert400(resp)  
 self.assertEqual(message, expected\_message)  
  
 complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 0)

We create a complainer and generate a token against it.

Then we will assure there are any complaints in the db

complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 0)

Then we will make a for loop and each time we will copy the dictionary (because it is a referenced type) and remove a key from it. We will make a request and each time we will expect BadRequest 400 against the field that we are removing from the dictionary for this iteration.

At the end of the for loop we are making sure that there are still no complaints in the database, because none of the requests succeeded.

Now we will make another method for the case where we are actually creating a complaint:

@patch(**"uuid.uuid4"**, mock\_uuid)

@patch.object(ComplaintManager, **"issue\_transaction"**, return\_value=None)  
@patch.object(S3Service, **"upload\_photo"**, return\_value=**"some.s3.url"**)  
def test\_complaint(self, mocked\_upload, mock\_transaction):  
 comp = ComplainerFactory()  
 token = generate\_token(comp)  
  
 complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 0)  
  
 data = {  
 **"title"**: **"Test"**,  
 **"description"**: **"Test test"**,  
 **"photo"**: encoded\_file,  
 **"photo\_extension"**: **"png"**,  
 **"amount"**: 10.00,  
 }  
 resp = self.client.post(  
 self.url,  
 data=json.dumps(data),  
 headers={  
 **"Content-Type"**: **"application/json"**,  
 **"Authorization"**: **f"Bearer** {token}**"**,  
 },  
 )  
  
 expected\_response = {  
 **"id"**: 1,  
 **"status"**: **"Pending"**,  
 **"amount"**: 10.0,  
 **"photo\_url"**: **"some.s3.url"**,  
 **"title"**: **"Test"**,  
 **"description"**: **"Test test"**,  
 }  
 self.assert200(resp)  
 resp = resp.json  
 resp.pop(**"create\_on"**)  
 self.assertEqual(resp, expected\_response)  
  
 name = mock\_uuid() + **"."** + data[**"photo\_extension"**]  
 path = os.path.join(TEMP\_FILE\_FOLDER, name)  
  
 complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 1)  
 self.assertEqual(complaints[0].id, resp[**"id"**])  
  
 mocked\_upload.assert\_called\_once\_with(  
 path, mock\_uuid(), data[**"photo\_extension"**]  
 )  
 mock\_transaction.assert\_called\_once\_with(  
 data[**"amount"**], **f"**{comp.first\_name} {comp.last\_name}**"**, comp.iban, complaints[0].id  
 )

This test is very complex – the definition of integration test – we will go through the overall process of creating. Generate complainer and token, assure there are no complaints yes. Make a request and define an expected response.

Here is the interesting part – when we are creating, we upload photo to s3 and also create a transaction in Wise.

When we are testing, we never test the external libraries – instead we test how our code is integrated with these libraries. To do that we using mocking (monkey patching in pytest).

The decorator

@patch.object(ComplaintManager, **"issue\_transaction"**, return\_value=None)

Allows us to patch the method issue\_transaction of the Complainer manager with some return value. More important you will notice that object patching is injecting the mocked object, to have access to it we describe a parameter in the method:

@patch.object(ComplaintManager, **"issue\_transaction"**, return\_value=None)  
@patch.object(S3Service, **"upload\_photo"**, return\_value=**"some.s3.url"**)  
def test\_complaint(self, mocked\_upload, mock\_transaction):

Mocked\_upload is the patched upload photo of our S3Service class and the mocked\_transaction is the patched method issue\_transaction of ComplaintManager class.

This way when we are testing, instead of going directly to s3 or wise we will tell our code to use the patched objects and we will never call s3 or wise while we are testing.

When we just use patch decorator it does not inject anything but it is mocking the selected object:

@patch(**"uuid.uuid4"**, mock\_uuid)

Now when we are trying to create an uuid, instead being random combination, for the tests it will always return the value of our function in base.py mock\_uuid.  
We pop the ‘create\_on’ from the response because in the model we are using .now form slqalchemy func and this could not be patched (if you are using now() from datetime you can read and patch with [freezegun](https://github.com/spulec/freezegun)).  
We can use the injections from patch.object to check if they are called and with what params:

mocked\_upload.assert\_called\_once\_with(  
 path, mock\_uuid(), data[**"photo\_extension"**]  
 )  
 mock\_transaction.assert\_called\_once\_with(  
 data[**"amount"**], **f"**{comp.first\_name} {comp.last\_name}**"**, comp.iban, complaints[0].id  
 )

At the end we need to make sure the complaint is stored in the database:

complaints = ComplaintModel.query.all()  
 self.assertEqual(len(complaints), 1)  
 self.assertEqual(complaints[0].id, resp[**"id"**])

**As a home assignment** write tests for what you can think of in complaint – getting complains, delete complaint as admin, approve/reject complaint as approver, test cases where exception could be raised and ect.

## Running tests in PyCharm

If you want to run the tests from the terminal you should go to the project root and just type

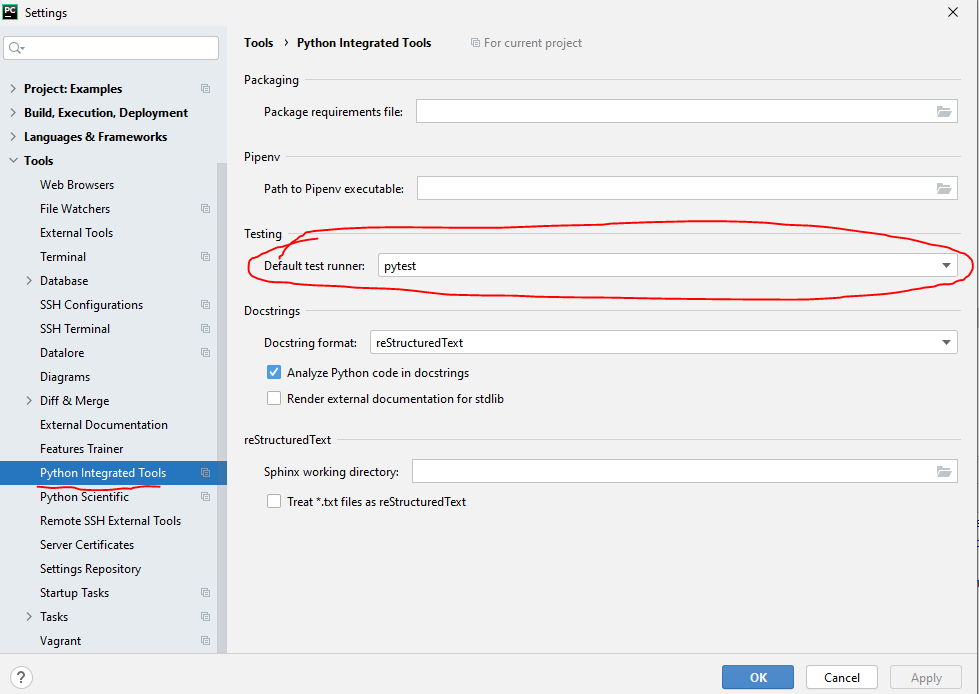
pytest

It will run all tests in folder **tests**

If you want to do the same thing from PyCharm, you need to setup test runner and configuration.

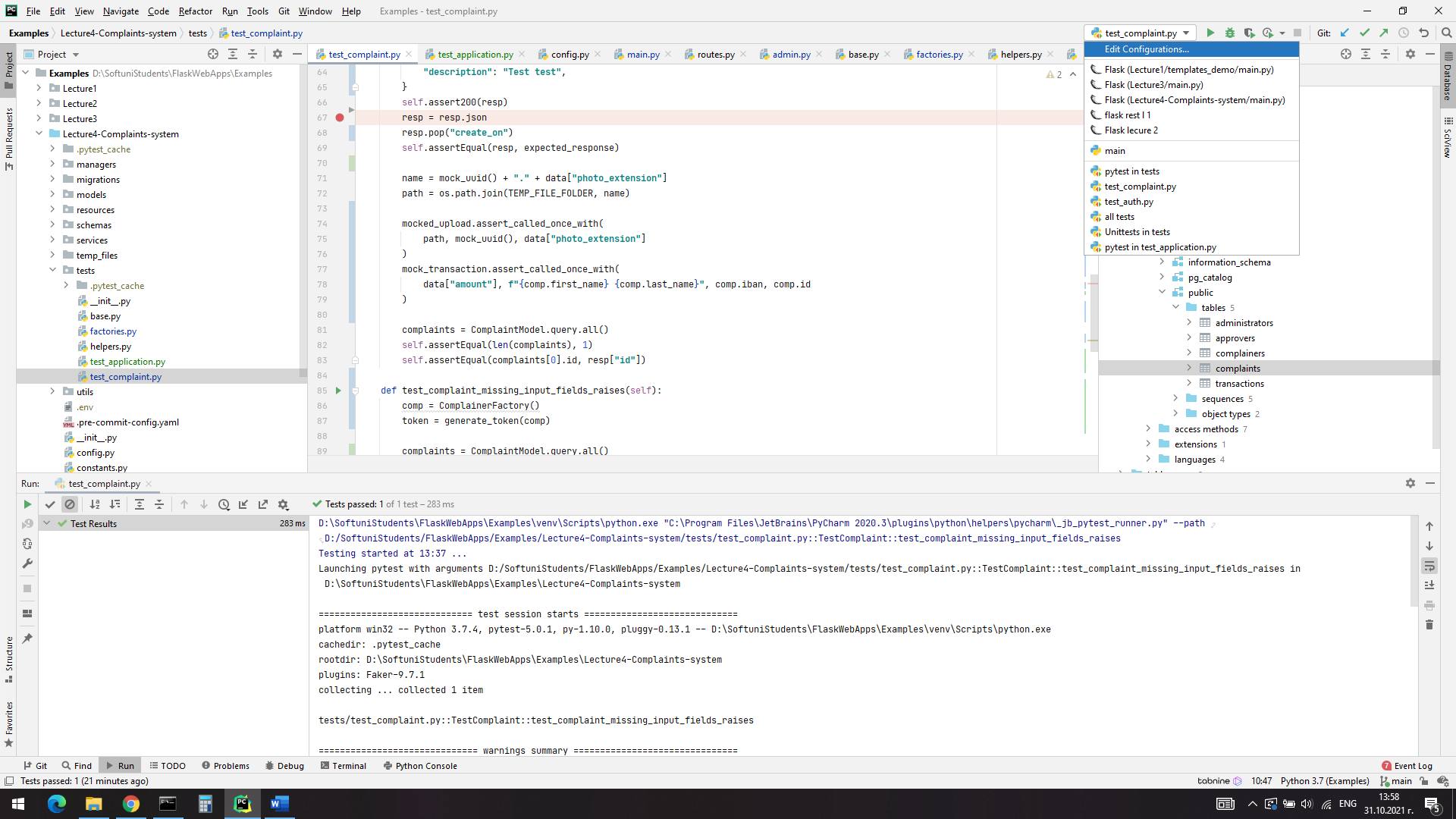
First change the default runner to pytest:

Go to **File => Settings => Tools => Python Integrated tools**

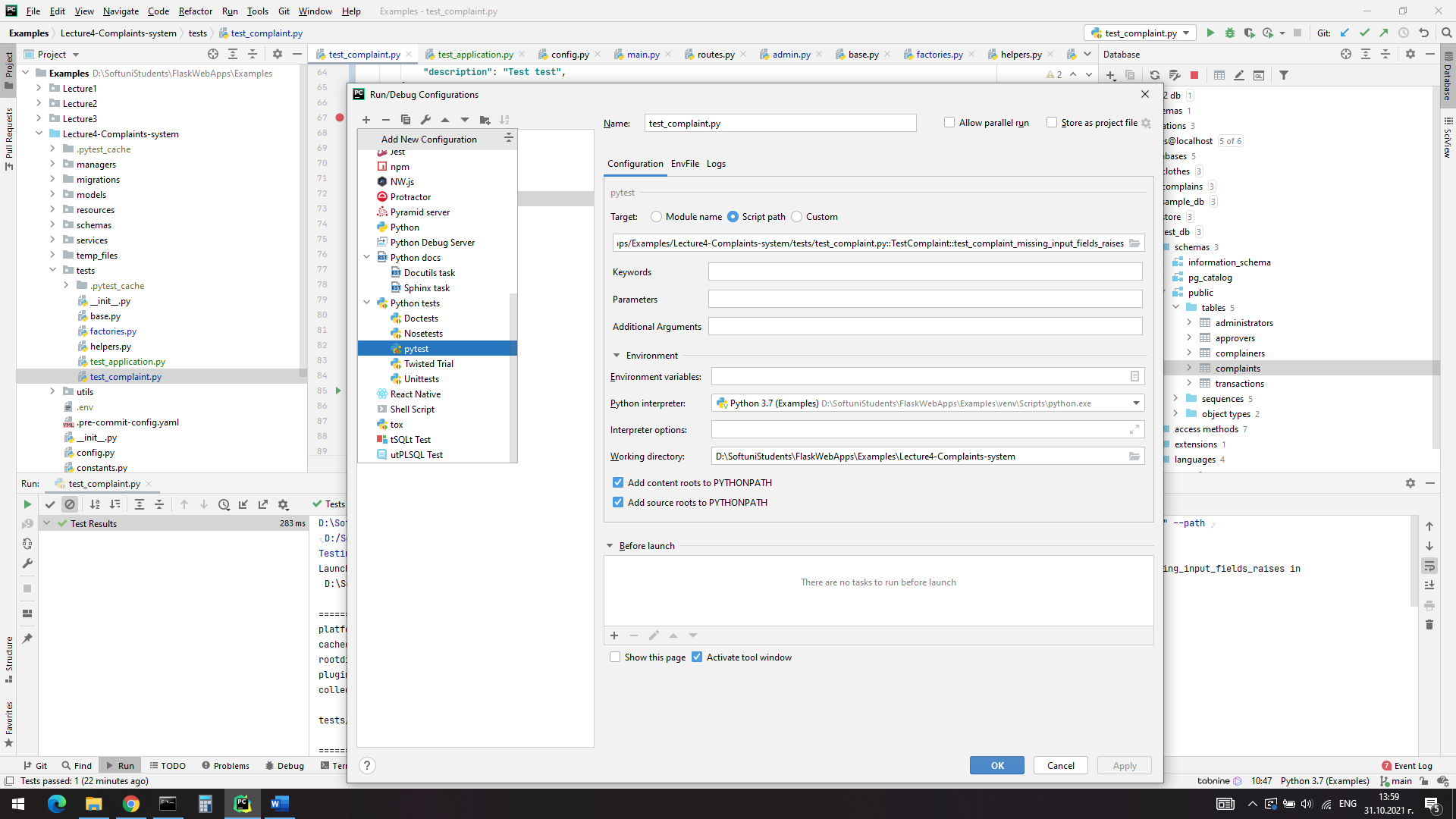


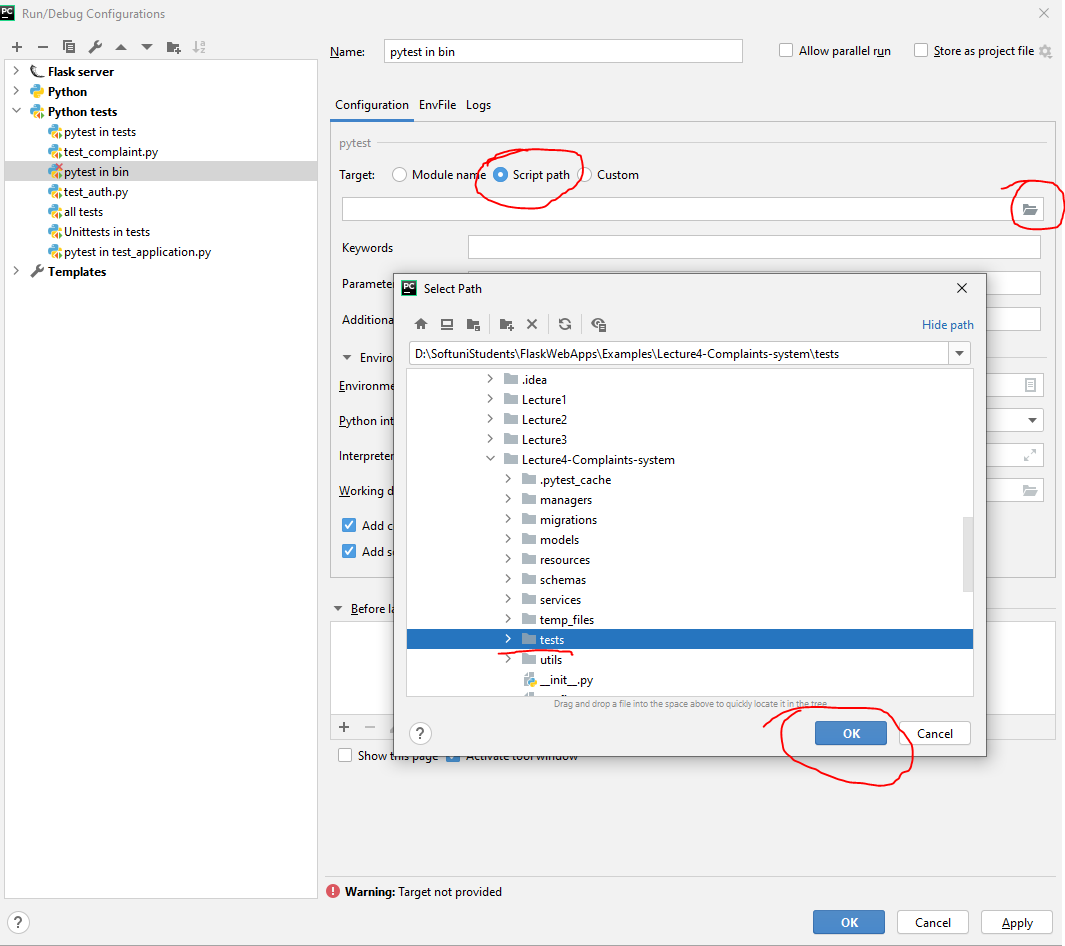
Then click ‘Apply’ and ‘ok’ and close the dialog.

Next go to configuration menu and click edit configuration:

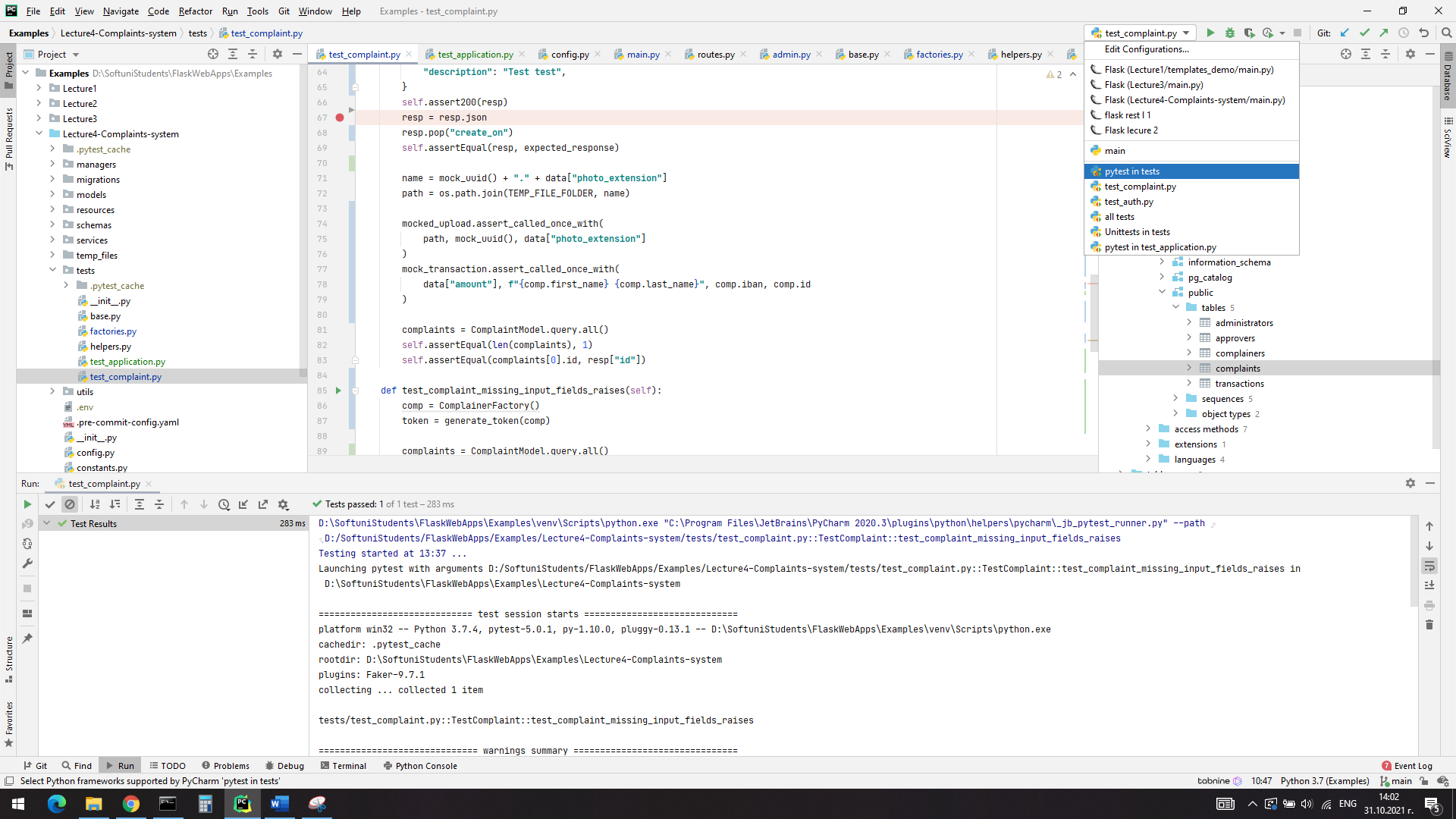


**From top left ‘+’ button select new pytest configuration:**





Select script path and then from the folder icon find your project and select tests folder. Click ok. Select the configuration and click the run button:



If you want to run a just a selected file. You need to do new configuration, but this time instead of selecting the tests folder, select the file inside this folder you want to run.

If you want to run a specific test method in specific class in specific file you need to nest the test class name and the method name in the script path like this:

{here your local path}/tests/test\_complaint.py**::**TestComplaint**::**test\_complaint\_missing\_input\_fields\_raises

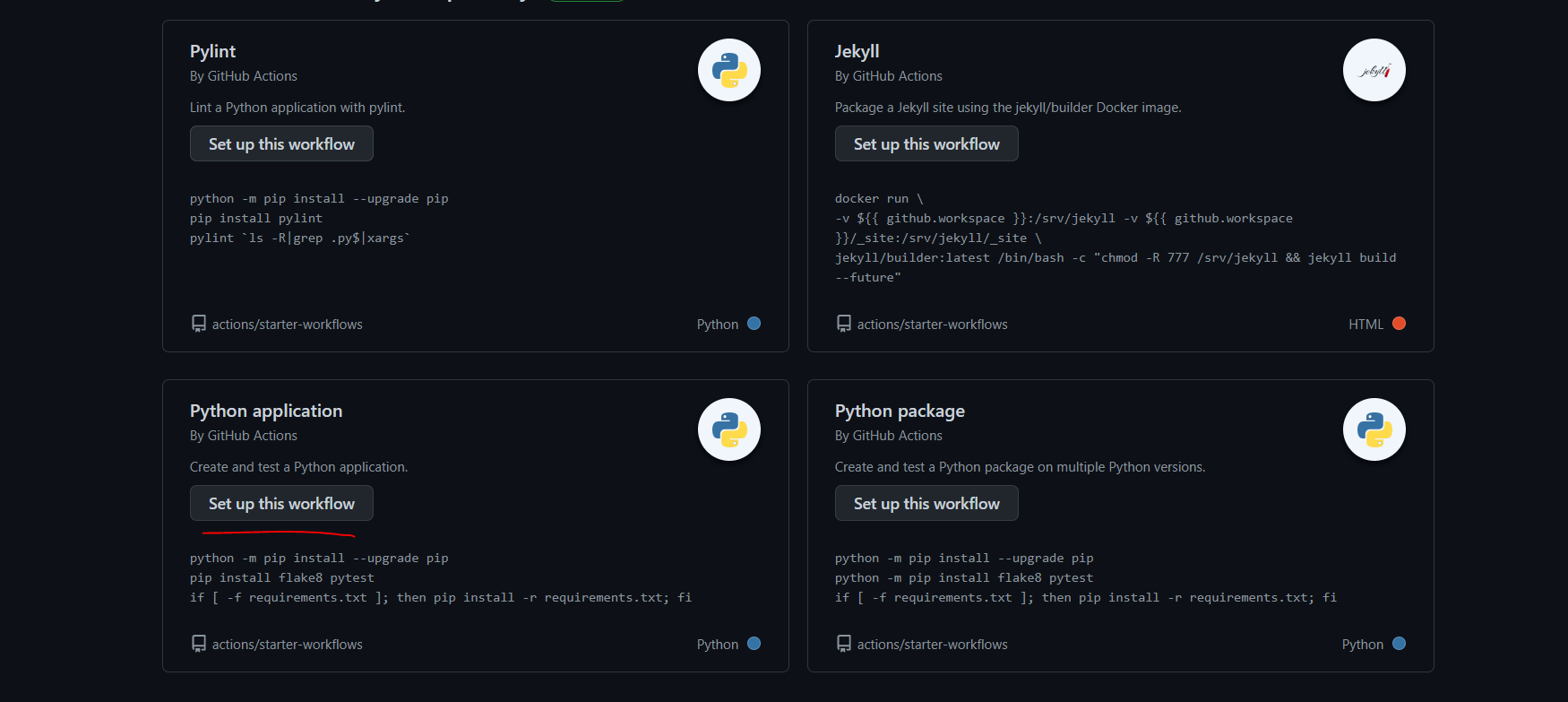
## BONUS: continues integration (CI) with GitHub actions

First, we need to add all packages that we are using in requirements.txt file – factory\_boy, flask\_testing, Flask\_cors and everything else you are using and it’s not added you should add with the respective versions.

If you want your tests to be run from GitHub when you open a PR or push a code to a specific branch, you can do it by logging to GitHub, selecting the repo and select Actions from the menu:



Search for python configuration and click ‘Set up this workflow’:



In the editor I am putting this code (please note that you need to replace the bold commands with your data):

name: Python complaint system

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

services:

postgres:

image: postgres:13.3

env:

POSTGRES\_USER**: {your user here}**

POSTGRES\_PASSWORD**: {your pass here}**

POSTGRES\_DB**: {your test db name here}**

ports:

**- 5432:5432**

# needed because the postgres container does not provide a healthcheck

options: --health-cmd pg\_isready --health-interval 10s --health-timeout 5s --health-retries 5

steps:

- uses: actions/checkout@v2

- name: Set up Python 3.7

uses: actions/setup-python@v2

with:

python-version: **"3.7"**

- name: Install dependencies and test

env:

**DB\_USER: “{your user}"**

**DB\_PASSWORD: “{your password}”**

**DB\_PORT: '5432'**

**DB\_NAME: 'test\_db'**

SECRET\_KEY: ""

AWS\_ACCESS\_KEY: ""

AWS\_SECRET: ""

AWS\_BUCKET: ""

AWS\_REGION: ""

WISE\_TOKEN: ""

WISE\_URL: ""

**POSTGRES\_PASSWORD: {your password}**

**POSTGRES\_USER: {your user}**

**POSTGRES\_DB: test\_db**

run: |

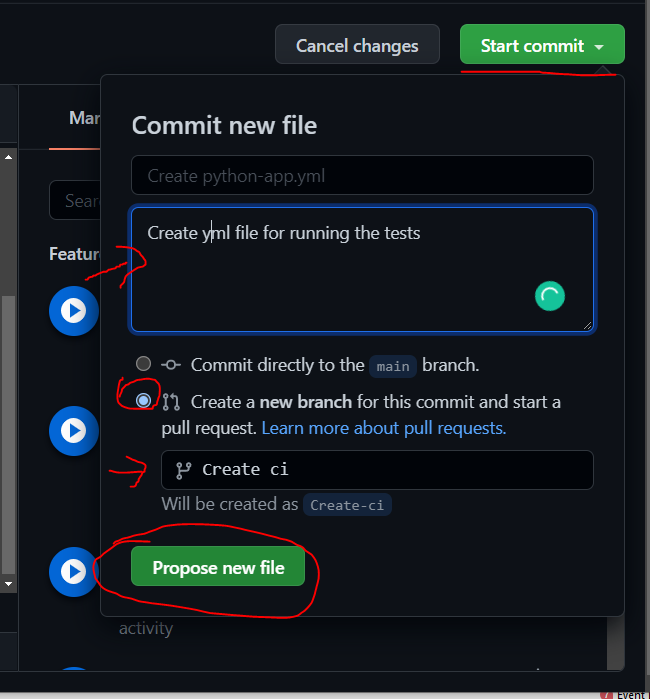
python -m pip install --upgrade pip

pip install -r requirements.txt

**export PGPASSWORD="{your password} "**

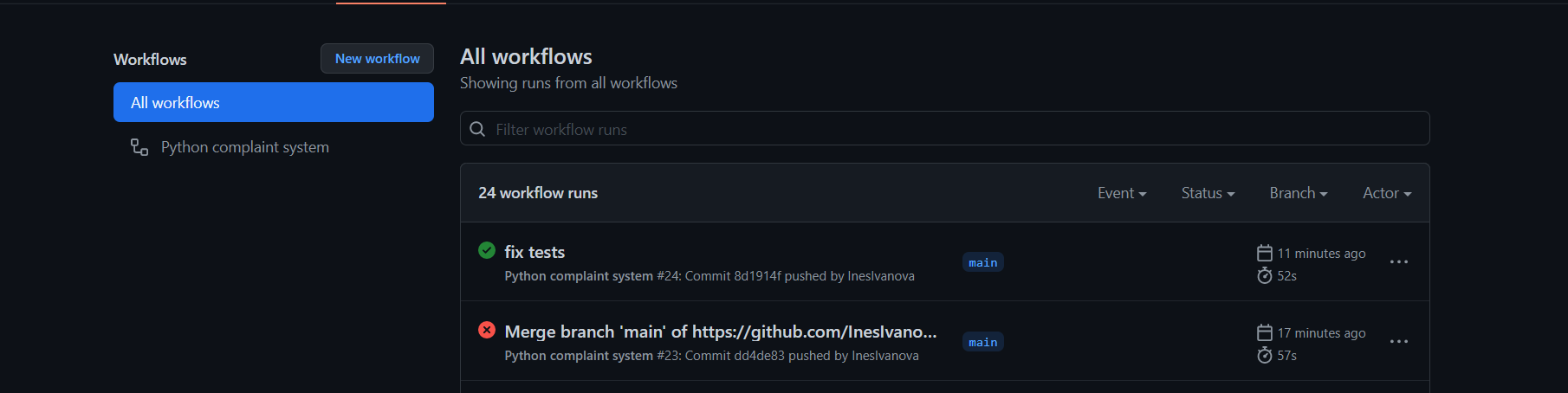
pytest

Then we will create a new branch against main:



Then we will go to the ‘Pull requests’ from the menu and we will merge it into main.

On the next push or PR opened against main, the ci will be triggered, if the tests are successful, we have a tick, otherwise X. You can monitor this from actions tab:



The project can be found [here](https://github.com/InesIvanova/Flask-course-prep-materials/tree/main/Lecture4-Complaints-system).