**[Python API test automation framework (Part 1) Introduction, Setup and Installation](https://automationhacks.io/2020/11/23/python-api-automation-framework-part1-python-and-dependencies-setup)**

**Ref:** https://automationhacks.io/2020/11/23/python-api-automation-framework-part1-python-and-dependencies-setup

Below is a non-exhaustive outline of what I have planned for this course at the moment:

* Chapter 1: Setup up python and virtualenv on mac or windows
* Chapter 2: Making HTTP requests
* Chapter 3: Adding fluent assertions
* Chapter 4: Working with XML
* Chapter 5: Working with JSON
* Chapter 6: API data/schema validation
* Chapter 7: Adding reporting
* Chapter 8: Running tests in parallel

This is also a chance for you to interact and provide ideas about which direction this course could take. Your perspective would be very valuable. Feel free to provide your comments using comment section below or DM me on twitter @automationhacks in case you would like to get something else added as well within this problem space.

## **Introduction to building API automation framework with Python**

Writing **Functional API automation** is a great way of getting fast and stable feedback about your system as well as exercise business flows and logic as compared to UI tests. In this course we would see the building blocks for a robust API automation framework using python to test your API’s.

We would cover automation development practices that you can follow and use different useful libraries in the python ecosystem to build our framework.

All the source code would be available on GitHub [course-api-framework-python](https://github.com/automationhacks/course-api-framework-python)

## **Setup and installations**

Before we begin building our framework for API testing, let’s make sure we have the basic dependencies already setup

Let’s dive in now

### **Setting up python and dependencies**

As a prerequisite to this tutorial, please ensure you have python3 installed on your machine. If not already installed, you can go to [python.org](https://www.python.org/downloads/) and download the latest version of python for your OS and run the installer

Alternatively, if you are on mac/linux and have homebrew/linuxbrew already installed then you can install python using below command

brew install python

Let’s test to make sure python is installed and available from the command line

python3 --version

Finally execute below to ensure you can access python REPL and you are all set

python3

Python 3.8.6 (default, Oct 21 2020, 11:06:14)

[Clang 11.0.3 (clang-1103.0.32.62)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>>

### **Setting up virtualenv**

With python setup. The next step that we need to take care of is to set up a virtualenv in which we can install all the required packages. You can just install modules directly on your base python installation but it’s an accepted best practice to use virtualenvs

I’ve already written a blog with details on how to you use pipenv to create a virtualenv for your project, please follow the same and setup pipenv and an empty env

you can refer to it [here](https://automationhacks.io/2020/07/12/how-to-manage-your-python-virtualenvs-with-pipenv/)

With this, we have our virtualenv ready for use.

### Install requests

To install any python module using pipenv you can use below command

pipenv install <module\_name>

Let’s install requests which we would use to actually make HTTP requests.

pipenv install requests

We can always check the package is installed by executing below (while inside the virtualenv)

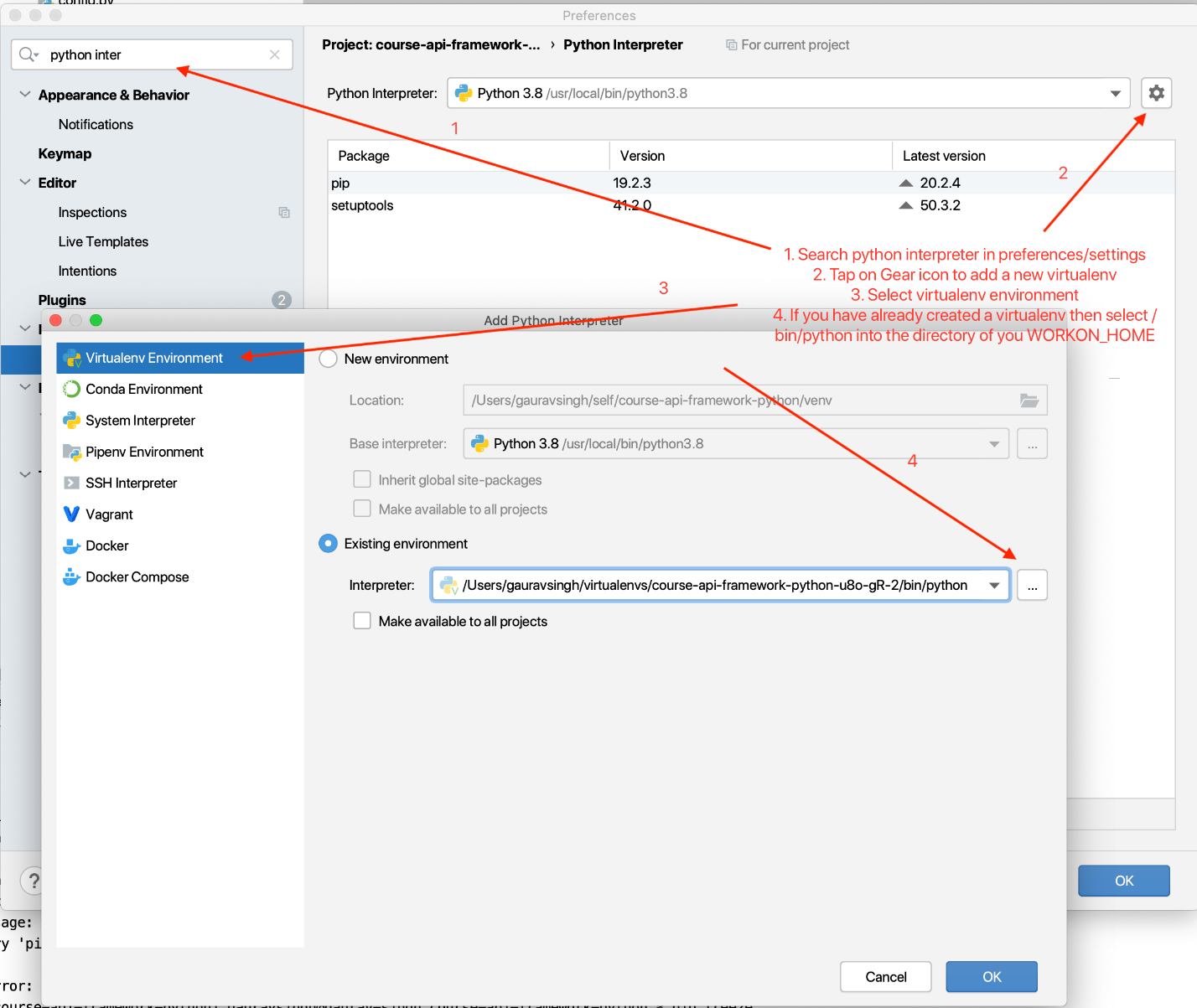
pip freeze

We will also use pytest as the test framework of choice

pipenv install pytest

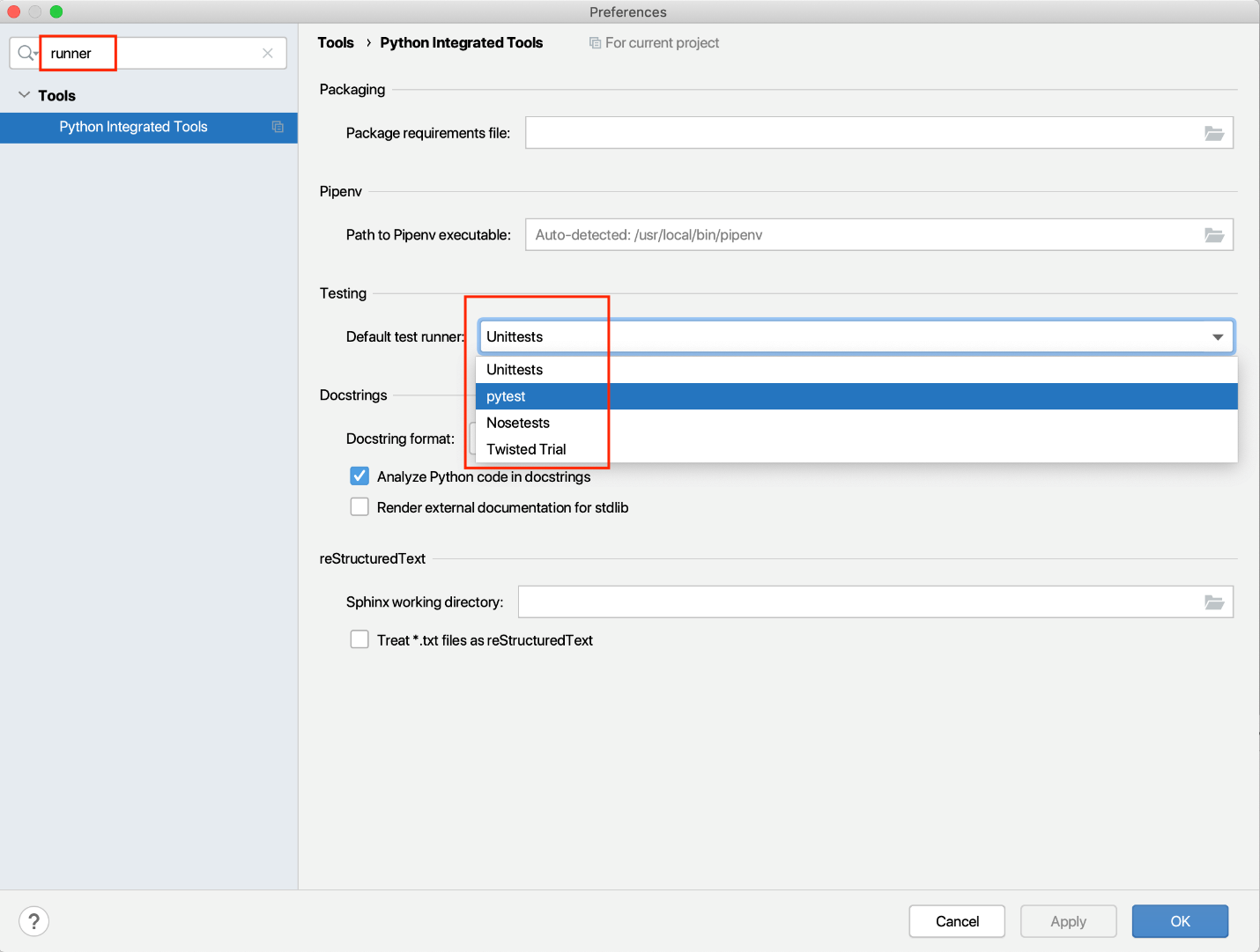
If you use pycharm as the editor of choice then you would need to select the newly created virtualenv as the project interpreter by following below steps

* Open Preferences/Settings
* Search for Python interpreter
* Click on Gear icon and then click on Add
* Select Existing environment
* Select path to bin/python for the created virtualenv (usually in WORKON\_HOME dir setup in your shell)



Also, let’s make sure pycharm knows that we intend to use pytest as the default test framework to run cases by going to

* Open preferences/settings
* Search runner
* Under Python integrated tools > Testing select default test runner as pytest



## Conclusion

And that all we need to get started with building our framework. We will add other modules when we discuss them in further chapters. Stay tuned for the next post on how to make an HTTP request using requests module

You can find the complete code for this course on Github at [automationhacks/course-api-framework-python](https://github.com/automationhacks/course-api-framework-python)

If you found this post useful, Do share it with a friend or colleague and if you have thoughts, I’d be more than happy to chat over at twitter or comments. Until next time. Happy Testing and Coding.

# [Python API test automation framework (Part 2) Making HTTP requests](https://automationhacks.io/2020/11/27/python-api-automation-framework-part2-making-http-requests)

## Understanding API under test

Let’s understand the API that we will be using in this tutorial a bit better.

We will be using [people-api](https://github.com/automationhacks/people-api) which is a set of CRUD HTTP operations developed using Python Flask, SQLAlchemy and uses sqlite as the database and represents a list of persons with first name, last name and an id

To setup clone people-api repo from github, cd to the newly cloned repo and then activate the pipenv by running below commands:

git clone git@github.com:automationhacks/course-api-framework-python.git

cd people-api

pipenv shell

Ensure all the dependencies are installed in the pipenv by executing:

pipenv install

Once done, Init the database by executing

python build\_database.py

This would seed the local sqlite database with some dummy records

Finally then start the HTTP service on your local by executing and ensuring this keeps on running during the test

python server.py

We can see from the log that the local service has started, you would also see logs for any request made to the server

\* Serving Flask app "config" (lazy loading)

\* Environment: production

WARNING: This is a development server. Do not use it in a production deployment.

Use a production WSGI server instead.

\* Debug mode: on

\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

\* Restarting with stat

\* Debugger is active!

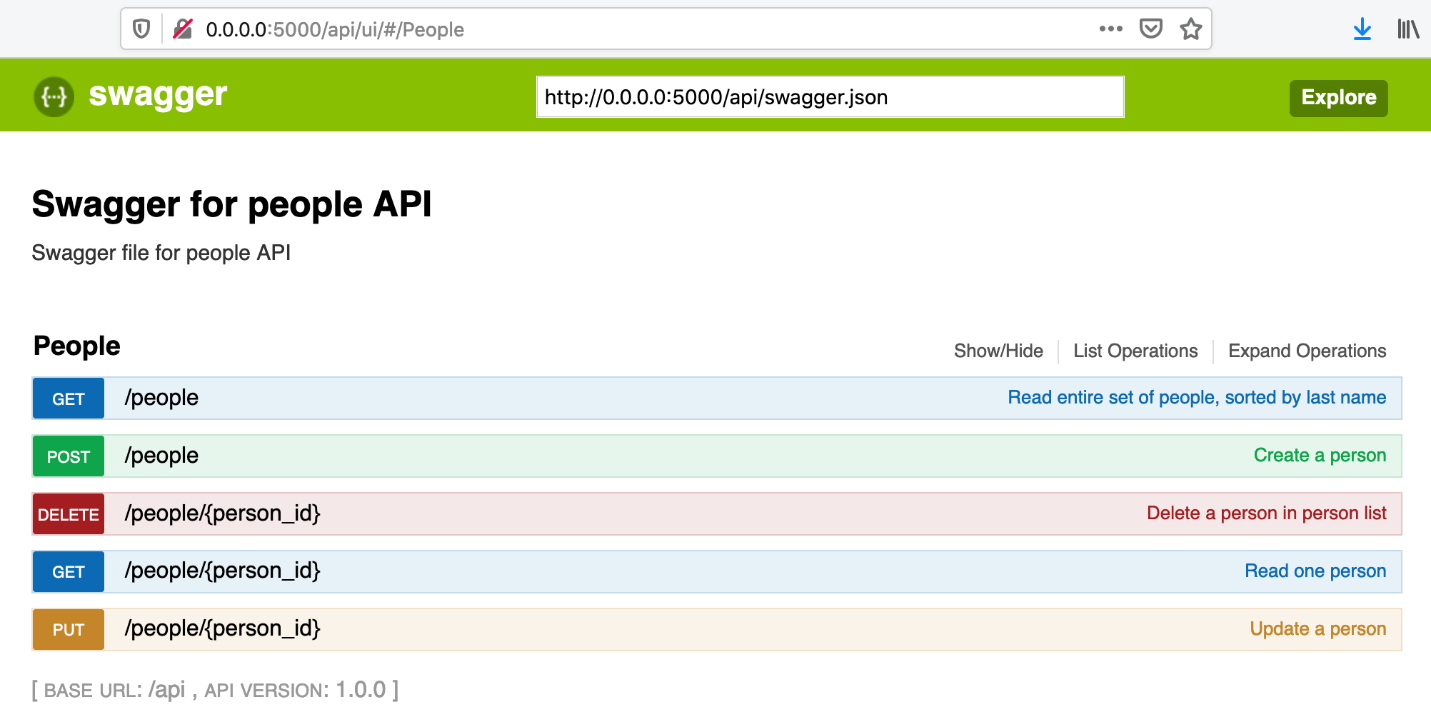
\* Debugger PIN: 105-008-664

Let’s navigate to the default host and port printed in flask logs http://0.0.0.0:5000 to see the application,

Note: To see the swagger spec for this API we can navigate to http://0.0.0.0:5000/api/ui/#/

Important Note 📕: On a windows machine, you may want to replace 0.0.0.0 with 127.0.0.1. Read [this post on How-To Geek](https://www.howtogeek.com/225487/what-is-the-difference-between-127.0.0.1-and-0.0.0.0/) to understand this is needed

As you can see this API supports basic CRUD operations (CREATE, READ, UPDATE, DELETE)



If you want to play around with these requests, I have also included a postman collection inside this repo under postman folder with all these basic operations already setup.

The API supports creating a person, updating him/her, deleting the person and able to read a user or all the users in the database. We will see these operations while writing tests.

## First test, read via GET API

Let’s see how we can create a basic test to make this HTTP request via python using requests module that we have already imported

The curl for this API is:

curl --location --request GET 'http://0.0.0.0:5000//api/people' \

--header 'Accept: application/json'

Below is the code snippet that wires up a basic HTTP GET request and then asserts that the service returns us a 200 status code and that the returned list of people has kent in it.

Notice we use assert\_that from assertpy module for fluent assertions, The failure messages from these are quite readable and we would discuss more about this in a future post. Please ensure you have that installed in your virtualenv using pipenv install assertpy

import requests

from assertpy.assertpy import assert\_that

from config import BASE\_URI

def test\_read\_all\_has\_kent():

# We use requests.get() with url to make a get request

response = requests.get(BASE\_URI)

# response from requests has many useful properties

# we can assert on the response status code

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

# We can get python dict as response by using .json() method

response\_text = response.json()

first\_names = [people['fname'] for people in response\_text]

assert\_that(first\_names).contains('Kent')

In above, we are using BASE\_URI variable to get the base api url to be used. This is stored in a config.py file as below:

BASE\_URI = 'http://0.0.0.0:5000/api/people'

## Creating a person using POST method

Alright, now that we have the basic case of GET API taken care of, let’s see how we can make a POST request.

Generally in Rest API’s when you need to create a resource you would use POST method

Below is the cURL to make the request, our python code is going to provide the same headers, body and URL to hit the request

curl --location --request POST 'http://0.0.0.0:5000//api/people' \

--header 'Content-Type: application/json' \

--header 'Accept: application/json' \

--data-raw '{

"fname": "TAU",

"lname": "User"

}'

Here is the complete flow that we want to automate:

* Create a new person in the db
* Verify if the newly created user is present using the GET API
* Filter to find if the created data is present in all the users in the system

|  |
| --- |
|  |
| def test\_new\_person\_can\_be\_added(): | |
|  | |

|  |
| --- |
| unique\_last\_name = create\_new\_person() |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # After user is created, we read all the users and then use list comprehension to find if the |
|  |

|  |
| --- |
| # created user is present in the response list |
|  |

|  |
| --- |
| peoples = requests.get(BASE\_URI).json() |
|  |

|  |
| --- |
| is\_new\_user\_created = search\_created\_user\_in(peoples, unique\_last\_name) |
|  |

|  |
| --- |
| assert\_that(is\_new\_user\_created).is\_not\_empty() |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def create\_new\_person(): |
|  |

|  |
| --- |
| # Ensure a user with a unique last name is created everytime the test runs |
|  |

|  |
| --- |
| # Note: json.dumps() is used to convert python dict to json string |
|  |

|  |
| --- |
| unique\_last\_name = f'User {str(uuid4())}' |
|  |

|  |
| --- |
| payload = dumps({ |
|  |

|  |
| --- |
| 'fname': 'New', |
|  |

|  |
| --- |
| 'lname': unique\_last\_name |
|  |

|  |
| --- |
| }) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Setting default headers to show that the client accepts json |
|  |

|  |
| --- |
| # And will send json in the headers |
|  |

|  |
| --- |
| headers = { |
|  |

|  |
| --- |
| 'Content-Type': 'application/json', |
|  |

|  |
| --- |
| 'Accept': 'application/json' |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # We use requests.post method with keyword params to make the request more readable |
|  |

|  |
| --- |
| response = requests.post(url=BASE\_URI, data=payload, headers=headers) |
|  |

|  |
| --- |
| assert\_that(response.status\_code, description='Person not created').is\_equal\_to(requests.codes.no\_content) |
|  |

|  |
| --- |
| return unique\_last\_name |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def search\_created\_user\_in(peoples, last\_name): |
|  |

|  |
| --- |
| return [person for person in peoples if person['lname'] == last\_name] |

[view raw](https://gist.github.com/automationhacks/3de97538503839ed2d681f4b63d1aef6/raw/7241a5e9dbf8109d74bcd5214f1293a75352c176/people_test.py) [people\_test.py](https://gist.github.com/automationhacks/3de97538503839ed2d681f4b63d1aef6#file-people_test-py) hosted with ❤ by [GitHub](https://github.com)

Here Ln 2: calls create\_new\_person() where we first create the body to be passed as part of the request

Notice on Ln 14 we are using dumps() method from json module to convert python dict to json string (serialization) and using str(uuid4()) to get a unique last name on Ln 13 to ensure we don’t have conflicting data on every test run

Finally we make the POST request using

response = requests.post(url=BASE\_URI, data=payload, headers=headers)

And then use search\_created\_user\_in() method to see if the newly created user is present in the database

## Deleting a person using DELETE method

How would we go about the delete method?

In this API we can delete a person by passing in the person\_id, it is always a good practice to create your own test data before trying delete operations instead of using any existing data since that could result in non-deterministic tests

So here it the flow:

* Create a new person
* Get the person id using read API
* Hit the delete endpoint and assert

Since we already have methods to create and then read users, the only thing left is to implement the DELETE operation by ensuring the person\_id is passed in the API path params, We use the fantastic f-strings to create our URL and then hit delete using requests.delete() method.

def test\_created\_person\_can\_be\_deleted():

persons\_last\_name = create\_new\_person()

peoples = requests.get(BASE\_URI).json()

newly\_created\_user = search\_created\_user\_in(peoples, persons\_last\_name)[0]

delete\_url = f'{BASE\_URI}/{newly\_created\_user["person\_id"]}'

response = requests.delete(delete\_url)

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

## Bonus

Note in assertions we are not specifying numeric value for the status codes, requests provides a lookup dictionary wherein you can easily specify the code using readable english like syntax and this is definitely better than parsing what code X means on [MDN - Mozilla developer network](https://developer.mozilla.org/en-US/docs/Web/HTTP) every time 😉

In other words instead of 200 we could write it as requests.codes.ok

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

## Conclusion

This is just a taste of what requests as a library can do. For more details on what all features it provides please read [requests docs](https://requests.readthedocs.io/en/master/user/quickstart/) and I hope this gives you a good idea on how to use requests to make HTTP calls. In the next post we would discuss more on working with JSON/XML data types and assertions. Stay tuned.

# [Python API test automation framework (Part 3) Adding fluent assertions using assertpy](https://automationhacks.io/2020/12/07/python-api-automation-framework-part3-fluent-assertions-using-assertpy)

## Starting with fluent assertions

How to start with fluent assertions with assertpy, you may ask?

Like any other python module, You should start by adding it to your virtual env by using below

pipenv install assertpy

Also, reading the exhaustive and well written Github readme is an excellent starting point as it provides you with a huge list of operations and methods that are possible with assertpy. You can learn more on [assertpy/assertpy](https://github.com/assertpy/assertpy)

## Basic assertions

Let’s see some basic assertions for our use cases and I would leave the rest to you to explore as and when you encounter certain use cases.

It’s generally always a good idea to skim through the library’s capabilities so that your brain remembers if capability **X** is something that you have already seen and just that act can help you with future use case

Let’s say you want to chain multiple assertions on a particular person from the same people response that we have been looking at:

[

{

"fname": "Kent",

"lname": "Brockman",

"person\_id": 2,

"timestamp": "2020-12-01T16:50:36.843495"

},

{

"fname": "Bunny",

"lname": "Easter",

"person\_id": 3,

"timestamp": "2020-12-01T16:50:36.843706"

}

...

]

Remember we did the check of extracting the first names from available persons and then added a check to see if Kent is contained in this:

first\_names = [people['fname'] for people in response\_text]

assert\_that(first\_names).contains('Kent')

We can use capabilities offered by assertpy to extract a list from a **list** of **dict** objects based on a certain key and then chain **multiple assertions** together. The above can be effectively replaced with.

assert\_that(response\_content).extracting('fname').is\_not\_empty().contains('Gaurav')

Observe that we are able to do multiple checks on the same object and this reads fluently like an english sentence. 🙌🏼

Also, Notice when the assertion fails, we get a friendly human readable error message like below:

AssertionError: Expected <['Kent', 'Bunny', 'Doug']> to contain item <Gaurav>, but did not.

Replace Gaurav with Kent to see the assertion pass.

The most common operations that you might perform with the assertion library are:

is\_equal\_to, is\_empty, contains, does\_not\_contain, matches

And these among many others are easily supported by assertpy out of the box.

## Printing a custom assertion message

Let’s say you want your own custom message to be printed when an assertion fails, You can easily do that use description='' keyword argument

assert\_that(response.status\_code, description='Person not created').is\_equal\_to(requests.codes.ok)

For example, I’ve wrongly modified the expected response code from no\_content to ok and assertpy gives the below readable stack trace along with our provided error message

> assert\_that(response.status\_code, description='Person not created').is\_equal\_to(requests.codes.ok)

E AssertionError: [Person not created] Expected <204> to be equal to <200>, but was not.

people\_test.py:63: AssertionError

## Soft assertions

You can also do **Soft assertions** (i.e not stop your test on the first failure) but instead raise a single assertion error at the end, to do so, ensure that you make use of soft\_assertions() function and python context manager using with clause

def test\_read\_all\_has\_kent():

# We use requests.get() with url to make a get request

response = requests.get(BASE\_URI)

# response from requests has many useful properties

# we can assert on the response status code

with soft\_assertions():

assert\_that(response.status\_code).is\_equal\_to(requests.codes.no\_content) # fails

# We can get python dict as response by using .json() method

response\_content = response.json()

# Use assertpy's fluent assertions to extract all fnames and then see the result is non empty and has

# Kent in it.

assert\_that(response\_content).extracting('fname').is\_not\_empty().does\_not\_contain('Kent') # fails

In the above example, we’ve deliberately modified the status code assertion and the response assertion.

Below is how it looks like once the test has run, we see both assertion messages show up.

E AssertionError: soft assertion failures:

E 1. Expected <200> to be equal to <204>, but was not.

E 2. Expected <['Kent', 'Bunny', 'Doug', 'New', 'New', 'New']> to not contain item <Kent>, but did.

## Conclusion

I hope you have a better sense of how a dedicated fluent assertion library makes your framework much better and I hope you would use them going forwards. That’s it for this post. We shall see how to work with XML and JSON data types in the next posts in the series

# [Python API test automation framework (Part 4) Working with XML using python lxml](https://automationhacks.io/2020/12/17/python-api-automation-framework-part4-working-with-xml-using-lxml)

## Introducing lxml

To work with XML in python, we would make use of the popular and powerful [lxml library](https://lxml.de/index.html) which is very useful for dealing with XML and is a wrapper over C libraries like **libxml2** and **libxslt** while retaining the simplicity of a native Python API

Let’s get started.

To set up, Ensure you have installed it in your pipenv using:

pipenv install lxml

## An example

Let’s consider we have to verify the XML response from an API that returns (the infamous) Covid data for the day with an overall world summary and a country wide summary breakup.

I’ve created a dummy service called **covid\_tracker.py** which is a python flask API to return a canned and static XML response

To ensure the service is running, execute below commands

# cd to dir

cd people-api

# activate pipenv and ensure all dependencies are installed

pipenv shell

pipenv install

# Run the local flask service

python covid\_tracker/covid\_tracker.py

Below is the cURL for this:

curl --location --request GET 'http://localhost:3000/api/v1/summary/latest'

And this would return a response like below:

<?xml version="1.0" encoding="UTF-8" ?>

<root>

<status>200</status>

<type>stack</type>

<data>

<summary>

<total\_cases>69169558</total\_cases>

<active\_cases>19895522</active\_cases>

<deaths>1574941</deaths>

<recovered>47699103</recovered>

<critical>104419</critical>

<tested>1003760026</tested>

<death\_ratio>0.022769279514551762</death\_ratio>

<recovery\_ratio>0.6895967587359746</recovery\_ratio>

</summary>

<change>

<total\_cases>653173</total\_cases>

<active\_cases>142334</active\_cases>

<deaths>12042</deaths>

<recovered>498799</recovered>

<critical>164</critical>

<tested>13146244</tested>

<death\_ratio>-0.00004130805512226471</death\_ratio>

<recovery\_ratio>0.0007060065458233122</recovery\_ratio>

</change>

<generated\_on>1607547603</generated\_on>

<regions>

<usa>

<name>USA</name>

<iso3166a2>US</iso3166a2>

<iso3166a3>USA</iso3166a3>

<iso3166numeric></iso3166numeric>

<total\_cases>15740193</total\_cases>

<active\_cases>6277786</active\_cases>

<deaths>295403</deaths>

<recovered>9167004</recovered>

<critical>26975</critical>

<tested>212565283</tested>

<death\_ratio>0.018767431886000382</death\_ratio>

<recovery\_ratio>0.5823946377277585</recovery\_ratio>

<change>

<total\_cases>222261</total\_cases>

<active\_cases>84705</active\_cases>

<deaths>2828</deaths>

<recovered>134728</recovered>

<death\_ratio>-0.00008656231889753868</death\_ratio>

<recovery\_ratio>0.0003405341268405415</recovery\_ratio>

</change>

</usa>

Let’s say, hypothetically we want to check that this API returns a valid no greater than a million of total worldwide cases and write a test for this.

Below is a test that achieves this.

import requests

from assertpy import assert\_that

from lxml import etree

from config import COVID\_TRACKER\_HOST

from utils.print\_helpers import pretty\_print

def test\_covid\_cases\_have\_crossed\_a\_million():

response = requests.get(f'{COVID\_TRACKER\_HOST}/api/v1/summary/latest')

pretty\_print(response.headers)

response\_xml = response.text

xml\_tree = etree.fromstring(bytes(response\_xml, encoding='utf8'))

# use .xpath on xml\_tree object to evaluate the expression

total\_cases = xml\_tree.xpath("//data/summary/total\_cases")[0].text

assert\_that(int(total\_cases)).is\_greater\_than(1000000)

Let’s break it down and understand whats happening here.

We make an HTTP Get call to our GET API /api/v1/summary/latest and get the response XML in text format.

response = requests.get(f'{COVID\_TRACKER\_HOST}/api/v1/summary/latest')

response\_xml = response.text

Next, to make use of this XML response, we need to deserialize (i.e. string to python object) it into a ElementTree object

Element tree belongs to the lxml library.

This can be done with below:

tree = etree.fromstring(bytes(response\_xml, encoding='utf8'))

📝 Its important to provide fromstring() data in bytes format with UTF-8 encoding since without that it would give error like: ValueError: Unicode strings with encoding declaration are not supported. Please use bytes input or XML fragments without declaration.

tree is now an object representation of the XML string and we can then use node.xpath('<your\_xpath\_expression>') to get the required node which we want to process.

In our current case we want the total\_cases node under the summary section.

We can get that using relative XPath expression as follows. If you are unfamiliar with XPath syntax, you refer to [this tutorial on w3schools.com](https://www.w3schools.com/xml/xpath_syntax.asp)

total\_cases = tree.xpath("//data/summary/total\_cases")[0].text

To get the text in the first node we use [0].text property

And finally now that we have the desired node, we could assert as follows

assert\_that(int(total\_cases)).is\_greater\_than(1000000)

## Another way to work with XPath using lxml

There is another way to make use of XPath wherein you can specify the expression upfront and then use it as needed.

Let’s say we want to assert that the total cases worldwide is greater than the total of cases across countries.

Below is the test, we could write for this:

def test\_overall\_covid\_cases\_match\_sum\_of\_total\_cases\_by\_country():

response = requests.get(f'{COVID\_TRACKER\_HOST}/api/v1/summary/latest')

pretty\_print(response.headers)

response\_xml = response.text

xml\_tree = etree.fromstring(bytes(response\_xml, encoding='utf8'))

overall\_cases = int(xml\_tree.xpath("//data/summary/total\_cases")[0].text)

# Another way to specify XPath first and then use to evaluate

# on an XML tree

search\_for = etree.XPath("//data//regions//total\_cases")

cases\_by\_country = 0

for region in search\_for(xml\_tree):

cases\_by\_country += int(region.text)

assert\_that(overall\_cases).is\_greater\_than(cases\_by\_country)

First few lines should be familiar now, Notice we use:

search\_for = etree.XPath("//data//regions//total\_cases")

Which gives us an XPath object but **does not evaluate** it as that point itself.

We make use of it to get a list of elements from the XPath and then use a loop to get the total for that specific region

cases\_by\_country = 0

for region in search\_for(xml\_tree):

cases\_by\_country += int(region.text)

And finally we can assert:

assert\_that(overall\_cases).is\_greater\_than(cases\_by\_country)

When I run this test, I can see it fail:

> assert\_that(overall\_cases).is\_greater\_than(cases\_by\_country)

E AssertionError: Expected <69169558> to be greater than <69822731>, but was not.

Which means that there is data mismatch bug 🐛 in this data set

## Conclusion

There are many other use cases which the lxml library can support. Discussing these here would result in a very long post. I would encourage you to get into the very well written [lxml docs](https://lxml.de/xpathxslt.html#xpath) when in doubt, for more details on your specific use cases.

You can find the complete code for this course on Github at [automationhacks/course-api-framework-python](https://github.com/automationhacks/course-api-framework-python)

# [Python API test automation framework (Part 5) Working with JSON and JsonPath](https://automationhacks.io/2020/12/25/python-api-automation-framework-part5-working-with-json)

efore we proceed, Let’s see some quick definitions that often come up when dealing with JSON format for API automation

* The process of encoding a **Python object** to **JSON** is called **serialization**
* The process of converting a **JSON** to a **Python object** is called **de-serialization**

I know these terms appear similar and sometimes confusing, however, don’t worry you’ll get the hang of it as you work with these more.

## Working with JSON

Python standard lib comes with out of the box support for JSON with the JSON module

There are primarily a couple of use cases that are often encountered

* Convert a python dict to JSON format to pass in the request
  + use json.dump() if you want to write to a file
  + or json.dumps() if you want to write to a python string
* Convert JSON to python dict
  + use json.load() if you want to read from a file directly
  + or json.loads() if you want to read from a python string

We’ve already seen the json.dumps() method in action in Chapter 2 and hence would not be going into detail.

## Understanding a typical API test flow

Let’s say we want to automate the below scenario on our people api

* Read JSON from a file
  + (This could be useful if you want to store the request body as a template somewhere instead of having it specified in the tests or even a test data file)
* Modify some parameter in the request
* Convert the python dict into a JSON string
* Pass the JSON payload to the POST request to create a user using the people-api
* Get all the users in the current database using GET api
* Assert that the new user is created in the system using the JSON path instead of manual parsing

I’ve gone ahead and created a test for this. Let’s understand the different pieces:

### tests/data/create\_person.json

{

"fname": "Sample firstname",

"lname": "Sample lastname"

}}

Firstly, we have the create\_person.json file under the **tests/data** directory to represent a sample request body (often called request payload as well).

This is in general a good pattern to follow since this avoids you having to mention request bodies explicitly in your tests and also makes your test files less bloated if you have a larger payload.

### utils/file\_reader.py

import json

from pathlib import Path

BASE\_PATH = Path.cwd().joinpath('..', 'tests', 'data')

def read\_file(file\_name):

path = get\_file\_with\_json\_extension(file\_name)

with path.open(mode='r') as f:

return json.load(f)

def get\_file\_with\_json\_extension(file\_name):

if '.json' in file\_name:

path = BASE\_PATH.joinpath(file\_name)

else:

path = BASE\_PATH.joinpath(f'{file\_name}.json')

return path

Next, we have utils/file\_reader.py to give us a function that can accept a file name in the tests/data directory, read it, and then send the JSON string back.

Couple of things to note here:

with path.open(mode='r') as f:

return json.load(f)

* Notice we are using the path.open instead of using pythons open method directly. This makes use of Path class from the [pathlib module](https://docs.python.org/3/library/pathlib.html) to easily help us build a file path (which is cross-platform out of the box) and use it easily.
* We also have a get\_file\_with\_json\_extension method which adds a .json extension if the file does not already have one.
* Also, we use json.load() and give it a file to read from directly and return a python object that we could

Alright, so this helps us get a python object.

### tests/people\_test.py

Here is how we can use this in our test.

Below is the complete test file.

I know it looks huge 😏 Let’s unpack the changes.

@pytest.fixture

def create\_data():

payload = read\_file('create\_person.json')

random\_no = random.randint(0, 1000)

last\_name = f'Olabini{random\_no}'

payload['lname'] = last\_name

yield payload

def test\_person\_can\_be\_added\_with\_a\_json\_template(create\_data):

create\_person\_with\_unique\_last\_name(create\_data)

response = requests.get(BASE\_URI)

peoples = loads(response.text)

# Get all last names for any object in the root array

# Here $ = root, [\*] represents any element in the array

# Read full syntax: https://pypi.org/project/jsonpath-ng/

jsonpath\_expr = parse("$.[\*].lname")

result = [match.value for match in jsonpath\_expr.find(peoples)]

expected\_last\_name = create\_data['lname']

assert\_that(result).contains(expected\_last\_name)

def create\_person\_with\_unique\_last\_name(body=None):

if body is None:

# Ensure a user with a unique last name is created everytime the test runs

# Note: json.dumps() is used to convert python dict to json string

unique\_last\_name = f'User {str(uuid4())}'

payload = dumps({

'fname': 'New',

'lname': unique\_last\_name

})

else:

unique\_last\_name = body['lname']

payload = dumps(body)

# Setting default headers to show that the client accepts json

# And will send json in the headers

headers = {

'Content-Type': 'application/json',

'Accept': 'application/json'

}

# We use requests.post method with keyword params to make the request more readable

response = requests.post(url=BASE\_URI, data=payload, headers=headers)

assert\_that(response.status\_code, description='Person not created').is\_equal\_to(requests.codes.no\_content)

return unique\_last\_name

#### Make use of pytest fixture for data setup

@pytest.fixture

def create\_data():

payload = read\_file('create\_person.json')

random\_no = random.randint(0, 1000)

last\_name = f'Olabini{random\_no}'

payload['lname'] = last\_name

yield payload

Above, instead of having the entire setup code in the test method. I’m making use of pytest fixtures to inject the data into the tests. Note the fixture named create\_data is passed as an argument to test method def test\_person\_can\_be\_added\_with\_a\_json\_template(create\_data):

We are getting the python dict as payload using read\_file('create\_person.json') and then using the [random module](https://docs.python.org/3/library/random.html) to generate a random no between 0 and 1000 and then adding it to a prefix.

Finally, we update that in the request body and then provide it to the test method using the yield keyword

We also modify the previously created create\_person\_with\_unique\_last\_name to optionally take the body in with a default value of None and use that to create JSON request body using the json.dumps() method or if not provided, still retain previous functionality of generating the request body.

## Using JSONPath

Finally, once our user is created let’s see how we can use a JSON path to extract values out of a JSON

# Get all last names for any object in the root array

# Here $ = root, [\*] represents any element in the array

# Read full syntax: https://pypi.org/project/jsonpath-ng/

jsonpath\_expr = parse("$.[\*].lname")

result = [match.value for match in jsonpath\_expr.find(peoples)]

expected\_last\_name = create\_data['lname']

assert\_that(result).contains(expected\_last\_name))

JSON path is a good way of working with a long nested JSON structure and it provides us XPath like capabilities. To add this library to our framework, add below:

pipenv install jsonpath-ng

For full details on the different use cases, this library can support refer to the PyPI page, [jsonpath-ng](https://pypi.org/project/jsonpath-ng/)

### An example

For our case, let’s say we want to perform the same action that we did earlier. i.e. get all the persons name and then check if the one that we expect is present inside the list.

We can specify the JSON path expression using the parse("$.[\*].lname") method

The above expression translates to:

* $ starting from the root,
* [\*] for any element inside the array
* .lname get the value of keys named lname

To get this JSON path to execute we call the find() method and give it the response JSON from the GET API response.

Finally, we assert that our expected last name is indeed present inside this list of users and fail if not found.

## Conclusion

In this chapter, we saw,

* How can we serialize or deserialize JSON
* Manipulate it
* and, finally get enhanced JSON parsing capabilities.

Understanding how these concepts would serve you well to form the foundation of a successful API test framework.

If you found this post useful, Do share it with a friend or colleague and if you have thoughts, I’d be more than happy to chat over at twitter or comments. Until next time. Happy Testing.

You can find the complete code for this course on Github at [automationhacks/course-api-framework-python](https://github.com/automationhacks/course-api-framework-python)

# [Python API test automation framework (Part 6) API response schema validation](https://automationhacks.io/2020/12/28/python-api-automation-framework-part6-api-response-schema-validation)

Most APIs return JSON that adhere to some contract set between its clients (could be another API or a web app etc). While you can write consumer-driven contract tests, sometimes you might want to just test with the live API and see if its response schema conforms to a fixed structure.

There are many different data validation libraries in the Python ecosystem and you can choose one that meets your objectives. We will use [Cerberus](https://docs.python-cerberus.org/en/stable/index.html) which is a quite popular library for this purpose. Other notable libraries are [jsonschema](https://pypi.org/project/jsonschema/), [voluptuous](https://pypi.org/project/voluptuous/) etc

## Setup

Install cerberus within your virtualenv

pipenv install cerberus

## Schema test for Read operation in People API

Let’s say we want to check that our people API’s response conforms to a schema that we expect:

Below is the structure we get when we hit the read API

{

"fname": "Doug",

"lname": "Farrell",

"person\_id": 1,

"timestamp": "2020-12-01T16:50:36.842997"

}

Cerberus works by defining a schema with all the fields inside the response object and their types and then validates if a sample response indeed met the schema need.

## Your first schema test

Below is a test to validate if Read one operation of people API meets a defined schema

import json

import requests

from cerberus import Validator

from config import BASE\_URI

schema = {

"fname": {'type': 'string'},

"lname": {'type': 'string'},

"person\_id": {'type': 'integer'},

"timestamp": {'type': 'string'}

}

def test\_read\_one\_operation\_has\_expected\_schema():

response = requests.get(f'{BASE\_URI}/1')

person = json.loads(response.text)

validator = Validator(schema)

is\_valid = validator.validate(person)

assert\_that(is\_valid, description=validator.errors).is\_true()

Let’s understand how this is constructed.

We start with defining the expected schema of the response object, Since it is a single object with certain keys like fname, lname etc, and values.

We can define a python dict with these schema details

"fname": {'type': 'string'}

Here for every field in the response, we specify key with the field name and value is another dict specifying the type like string, number, boolean, date etc.

See the full list of types on [cerberus docs](https://docs.python-cerberus.org/en/stable/validation-rules.html#type)

Sweet, Below is how the schema looks like for the read response

schema = {

"fname": {'type': 'string'},

"lname": {'type': 'string'},

"person\_id": {'type': 'integer'},

"timestamp": {'type': 'string'}

}

We then hit the GET API with an expected person id and then convert the response to a python dict using loads()

response = requests.get(f'{BASE\_URI}/1')

person = json.loads(response.text)

Note: Hard coding a user id like 1 (in the request URL) is often something to be avoided. You might want to create a new user and then do this test, however since this API gets seeded with some dummy data, we are following this approach for demo purposes only.

We then initialize an instance of Validator class with this schema. Optionally, if we want to specify that all the keys are required in this schema then we can add require\_all=True keyword argument. Or we could even specify this at a field level using 'required': True/False in the schema itself.

validator = Validator(schema, require\_all=True)

We can assess if the JSON matches this schema with below:

is\_valid = validator.validate(person)

And if we want to raise an assertion error if Cerberus finds a mismatch then we can print validator.errors

assert\_that(is\_valid, description=validator.errors).is\_true()

When we run the test for our current people API, we see the test passes.

To see how it would look like in case of a failure we can change the type of person\_id from number to string and that would raise the error message below, notice we get to know the field that is mismatched and what the validation failure is.

AssertionError: [{'person\_id': ['must be of string type']}] Expected <True>, but was not.

## Test for read all operation

How does this test look like for the Read all operation?

def test\_read\_all\_operation\_has\_expected\_schema():

response = requests.get(BASE\_URI)

persons = json.loads(response.text)

validator = Validator(schema, require\_all=True)

with soft\_assertions():

for person in persons:

is\_valid = validator.validate(person)

assert\_that(is\_valid, description=validator.errors).is\_true()

Essentially, We get the list of persons and then repeat the same validation for all the records in this list while wrapping it with a soft assertion to ensure all the validation failures are collected and printed in the end.

## Conclusion

Schema validation is an important component to include in your API automation framework and I hope you have a basic understanding of how to use a tool like Cerberus to achieve this. For understanding, all the nuances of this approach feel free to dig deep into Cerberus docs which lists a lot of the functionality that is available.

You can find the complete code for this course on Github at [automationhacks/course-api-framework-python](https://github.com/automationhacks/course-api-framework-python)

# [Python API test automation framework (Part 7) Refactoring structure](https://automationhacks.io/2021/01/08/python-api-automation-framework-part7-refactoring-structure)

So far, we’ve been observing how to achieve a specific piece of functionality in our API framework using available packages in the python ecosystem.

However, when we talk about a framework, we have to do a bit more upfront thinking about how this would be used by team members and the overall structure in which it is organized becomes as important as well.

One way that I typically approach is to decide on an structure based on initial needs and then continuously refactor as the domain, context changes, typically after a few cycles the structure and the relationships between different modules, classes becomes clear and you have your framework structure.

Let’s see what some of these changes are:

## Remove unnecessary comment and replace them with well named functions/classes

If you have been following along earlier commits/posts in this series, you would have seen that we had given inline comments using # above many lines of code.

While this might be okay for a demo framework like this, I still wanted to call out that this practice could be quite bad in a real project. Reason being, with every passing wave of refactoring you might extract certain functions or rename variables or even change behavior in some cases.

Quite often, it’s easy to forget updating the comment and then you have comments which don’t even match with underlying logic and that is just plain confusing for the person who reads your code. 🤷🏼‍♂️

I’ve thus removed these from this branch and from the final master branch as well.

Below is the updated tests file with this and some other changes:

tests/people\_test.py

import random

from json import loads

import pytest

import requests

from assertpy.assertpy import assert\_that

from jsonpath\_ng import parse

from clients.people.people\_client import PeopleClient

from config import BASE\_URI

from tests.helpers.people\_helpers import search\_created\_user\_in

from utils.file\_reader import read\_file

client = PeopleClient()

def test\_read\_all\_has\_kent():

response = client.read\_all\_persons()

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

assert\_that(response.as\_dict).extracting('fname').is\_not\_empty().contains('Kent')

def test\_new\_person\_can\_be\_added():

last\_name, response = client.create\_person()

peoples = client.read\_all\_persons().as\_dict

is\_new\_user\_created = search\_created\_user\_in(peoples, last\_name)

assert\_that(is\_new\_user\_created).is\_not\_empty()

Notice, we have replaced all the comments with better named functions that clarify intent.

## Introducing client class as an abstraction layer

In earlier posts, we were directly calling requests.get(), post() and other methods and working with response objects.

This is often a practice to avoid because it tightly couples our test implementation to request libraries implementation

While requests (or, any other library) today does a good job for us, we might have to tweak its behavior for all our tests or might have to switch to a different library altogether.

To ensure minimal changes to test code in those scenarios, it’s always a good idea to wrap these third party library code into an abstraction that we maintain.

One common pattern is to **abstract these domain operations into their own client class/module.**

I’ve thus introduced a people\_client.py

clients/people/people\_client.py

from json import dumps

from uuid import uuid4

from clients.people.base\_client import BaseClient

from config import BASE\_URI

from utils.request import APIRequest

class PeopleClient(BaseClient):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.base\_url = BASE\_URI

self.request = APIRequest()

def create\_person(self, body=None):

last\_name, response = self.\_\_create\_person\_with\_unique\_last\_name(body)

return last\_name, response

def \_\_create\_person\_with\_unique\_last\_name(self, body=None):

if body is None:

last\_name = f'User {str(uuid4())}'

payload = dumps({

'fname': 'New',

'lname': last\_name

})

else:

last\_name = body['lname']

payload = dumps(body)

response = self.request.post(self.base\_url, payload, self.headers)

return last\_name, response

def read\_one\_person\_by\_id(self, person\_id):

pass

def read\_all\_persons(self):

return self.request.get(self.base\_url)

def update\_person(self):

pass

def delete\_person(self, person\_id):

url = f'{BASE\_URI}/{person\_id}'

return self.request.delete(url)

Notice this has simple wrapper methods for the same operations that we were directly calling from the test function and also makes every call to requests from an object of APIRequest() class instead of calling request module directly

How does this APIRequest class look like?

utils/request.py

from dataclasses import dataclass

import requests

@dataclass

class Response:

status\_code: int

text: str

as\_dict: object

headers: dict

class APIRequest:

def get(self, url):

response = requests.get(url)

return self.\_\_get\_responses(response)

def post(self, url, payload, headers):

response = requests.post(url, data=payload, headers=headers)

return self.\_\_get\_responses(response)

def delete(self, url):

response = requests.delete(url)

return self.\_\_get\_responses(response)

def \_\_get\_responses(self, response):

status\_code = response.status\_code

text = response.text

try:

as\_dict = response.json()

except Exception:

as\_dict = {}

headers = response.headers

return Response(

status\_code, text, as\_dict, headers

)

Notice, we have created wrapper methods for the **HTTP operations** that our current framework needs, namely **post(), delete()** and also we are wrapping the response object into a custom object that we control.

This is very powerful for the above mentioned reasons

* We can now easily modify request libraries behavior without changing underlying tests
* Our clients are not depending on requests library and we are free to switch to a different implementation in the future if such a need arises.

We also make use of Pythons [@dataclass](https://realpython.com/python-data-classes/) annotation to create this data holder and use the private method \_\_get\_responses to take a response object from requests and then return our own implementation.

### Make the client thin

Another thing to observe is, we have not put any validations on **response status code, body, headers** in these client methods.

This is intentional since we want this to be the tests concern. The client just gives a simple class to make these HTTP requests and leaves all implementation details for different validations to the tests.

This approach ensures we can reuse the same method in the client to test for a **success (2XX) or a failure case (4XX, 5XX)** status code.

## Abstract helper methods into their own class/module

I’ve also moved some helper methods related to searching a person in response and the JSON path implementation into its own module, this ensures that the test code is clean and there is less cognitive overload on the person looking at the code to grasp each and every detail.

For instance a descriptive method name like search\_created\_user\_in is always going to be more understandable than [person for person in peoples if person['lname'] == last\_name][0] which requires the code reader to grasp python list comprehension, array slicing and what not.

tests/helpers/people\_helpers.py

from jsonpath\_ng import parse

def search\_created\_user\_in(peoples, last\_name):

return [person for person in peoples if person['lname'] == last\_name][0]

def search\_nodes\_using\_json\_path(peoples, json\_path):

jsonpath\_expr = parse(json\_path)

return [match.value for match in jsonpath\_expr.find(peoples)]

## Abstract common assertions into their own module

I’ve also extracted some assertions into their own functions and moved them into their own module, for above reasons

tests/assertions/people\_assertions.py

from assertpy import assert\_that

def assert\_people\_have\_person\_with\_first\_name(response, first\_name):

assert\_that(response.as\_dict).extracting('fname').is\_not\_empty().contains(first\_name)

def assert\_person\_is\_present(is\_new\_user\_created):

assert\_that(is\_new\_user\_created).is\_not\_empty()

## Extract fixtures into conftest.py file

Pytest framework has the flexibility to put [fixture code into a conftest.py](https://stackoverflow.com/questions/34466027/in-pytest-what-is-the-use-of-conftest-py-files) file which is auto discovered and allows us to separate setup teardown code from the actual tests.

These fixtures can be reused at different levels and you can easily move them up the directory structure if you want the same fixtures to be available to many different test files

tests/conftest.py

import random

import pytest

from utils.file\_reader import read\_file

@pytest.fixture

def create\_data():

payload = read\_file('create\_person.json')

random\_no = random.randint(0, 1000)

last\_name = f'Olabini{random\_no}'

payload['lname'] = last\_name

yield payload

In the end our entire test file is now well refactored and much more readable.

import requests

from clients.people.people\_client import PeopleClient

from tests.assertions.people\_assertions import \*

from tests.helpers.people\_helpers import \*

client = PeopleClient()

def test\_read\_all\_has\_kent():

response = client.read\_all\_persons()

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

assert\_people\_have\_person\_with\_first\_name(response, first\_name='Kent')

def test\_new\_person\_can\_be\_added():

last\_name, response = client.create\_person()

assert\_that(response.status\_code, description='Person not created').is\_equal\_to(requests.codes.no\_content)

peoples = client.read\_all\_persons().as\_dict

is\_new\_user\_created = search\_created\_user\_in(peoples, last\_name)

assert\_person\_is\_present(is\_new\_user\_created)

def test\_created\_person\_can\_be\_deleted():

persons\_last\_name, \_ = client.create\_person()

peoples = client.read\_all\_persons().as\_dict

new\_person\_id = search\_created\_user\_in(peoples, persons\_last\_name)['person\_id']

response = client.delete\_person(new\_person\_id)

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

def test\_person\_can\_be\_added\_with\_a\_json\_template(create\_data):

client.create\_person(create\_data)

response = client.read\_all\_persons()

peoples = response.as\_dict

result = search\_nodes\_using\_json\_path(peoples, json\_path="$.[\*].lname")

expected\_last\_name = create\_data['lname']

assert\_that(result).contains(expected\_last\_name)

## A Simple heuristic to follow: Make it work => beautiful => fast

Notice how we approached this whole exercise and did these refactoring in phases. That is one iterative way to build frameworks and has the benefit of ensuring only the required bits of code come into the framework.

Below is the general heuristic i follow which has been written about multiple times.

* **Make it work**: Write the simplest thing possible that could work and get the job done first, don’t do any [premature optimizations](http://wiki.c2.com/?PrematureOptimization) in this phase
* **Make it beautiful**: With implementation details figured out, think of how you can structure it better and refactor existing lines of code, methods into their own logical place. This is where you think of patterns, structure, which responsibilities lie with which class/module etc.
* **Make it fast**: Make the code you write something that runs really fast. In this phase, You optimize your implementation (after profiling current performance) and think about concurrency as well. More on this in a future post.

## Conclusion

In summary, Most of the concerns were abstracted into a dedicated place which can be easily understood and modified in the future.

# [Python API test automation framework (Part 8) Adding reporting with report portal](https://automationhacks.io/2021/02/03/python-api-automation-framework-part8-adding-reporting-with-report-portal)

Ah, Test results reporting is a favorite topic of discussion among test automation engineers.

While everyone wants to create their own reporting dashboard/site because it sure is fun to mess around with JS, HTML, CSS, Store test results in a relational or document database and maybe create some APIs routes using a web framework like Flask, Django.

It’s also true that all this is a lot of work to set up in the correct way. I won’t discourage you if you want to build the next-gen reporting solution that the community can use.

However, Should we really invest our valuable time in reinventing the wheel? Probably not, (at least, if we can avoid it 😉)

The test results reporting space has some great tools and libraries already available that get the job done however, of late one of my absolute favorites is the report portal because of the ton of features it packs out of the box

In this post, We will see how to integrate it into our framework that we’ve been building so far:

## Setup report portal using docker

I had previously written a blog on setting up a local ReportPortal instance on Docker and demonstrated a basic Java test on Gradle for reporting. You can read that post here: [How to setup ReportPortal on a local docker instance](https://automationhacks.io/2020/03/02/how-to-setup-reportportal-on-a-local-docker-instance/)

Essentially, in a nutshell, the setup is quite easy to do using docker as it pulls all the required dependencies and spins up an environment for your use very quickly

Below are the steps to follow to set up your containers

# Step 1: Download docker-compose YAML file

curl https://raw.githubusercontent.com/reportportal/reportportal/master/docker-compose.yml -o docker-compose.yml

# Step 2: Spin up containers

# This will download the images (one time) over the internet so ensure you are on a good connection

docker-compose -p reportportal up -d --force-recreate

Once the above command finishes, verify if all the containers are spun up by executing:

docker ps -a

Once done you should be able to open up the local site at http://localhost:8080/ui/ with default id and password in report portals website (at the time of writing this blog the credentials are user: superadmin, password: erebus)

(🙏🏼 : Please change this in any production instance that you wish to use in your organizations unless you want those pesky hackers to have access to your projects test results)

## Bug workaround: elastic search keeps on restarting:

You might notice that once you log in, you get pop-ups like “An error occurred” and your session is terminated.

To identify the root cause and fix this issue, follow the below steps.

If this works perfectly for you, then feel free to skip these steps and move on to the next section that talks about setting up our framework

docker ps -a

We can observe that part of the problem is that the elasticsearch container keeps on restarting.

7b6052df5d30 docker.elastic.co/elasticsearch/elasticsearch:7.3.0 "/usr/local/bin/dock…" 11 minutes ago Restarting (1) 53 seconds ago reportportal\_elasticsearch\_1

Yes, I know libraries have their own quirks 🤷🏼‍♂️, you can fix this quite easily by following the below steps:

# Stop all running containers

docker stop $(docker ps -aq)

# Create /data/elasticsearch directory

mkdir -p data/elasticsearch

# Give permissions

chmod 777 data/elasticsearch

# Change ownership of this directory

sudo chown 1000:1000 data/elasticsearch/

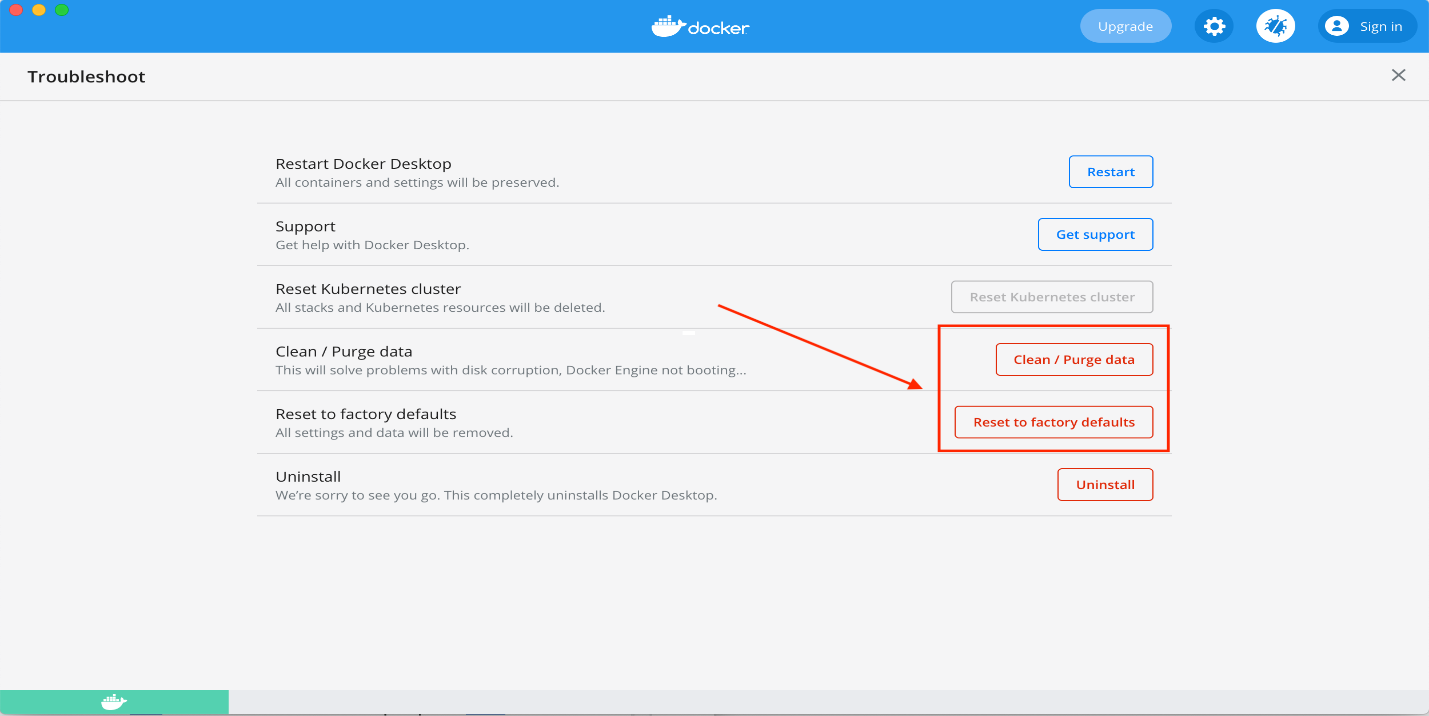
# Finally bring up the containers again

docker-compose -p reportportal up -d

You can even quit docker for mac and restart it once again.

This workaround is from a Github issue on the report portal website. You can see more details [here](https://github.com/reportportal/reportportal/issues/500#issuecomment-430941820?). I even started a thread on the [Report portal slack channel](https://reportportal.slack.com/archives/C2GTJTTNH/p1611587152043900) that you can follow for further discussions.

Note: If the above steps still result in elastic search restarting, you can manually reset docker by going to troubleshoot and click on Clean / Purge data and Reset to factory defaults



## Integrate with pytest

Let’s understand how we can integrate our test suite (run using pytest) to report results directly into report portal

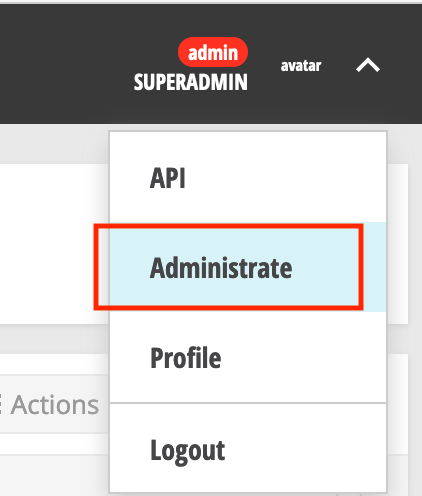
You can find more details about different types of test framework integrations

* Supported by report portal supports [here](https://reportportal.io/docs/Test-Framework-Integration)
* Complete documentation for the pytest plugin [here](https://github.com/reportportal/agent-python-pytest)

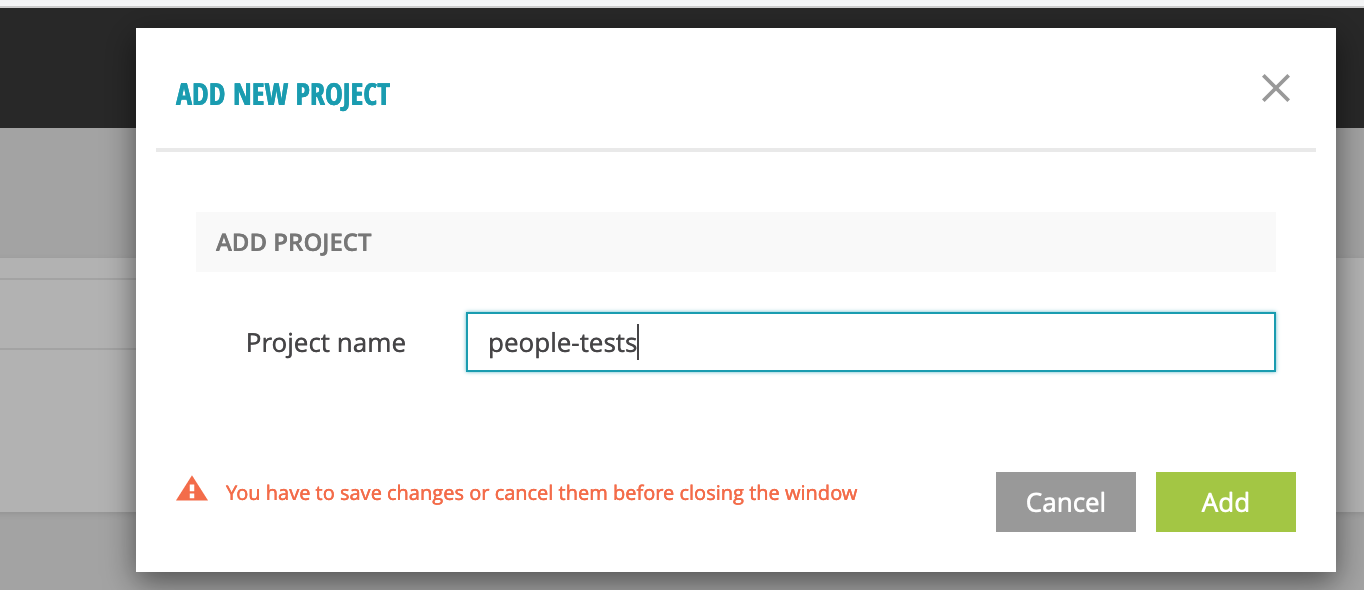
### Step 1: Create a project in report portal and update pytest.ini file

It is recommended to create a new project in report portal where you want to report test results to:

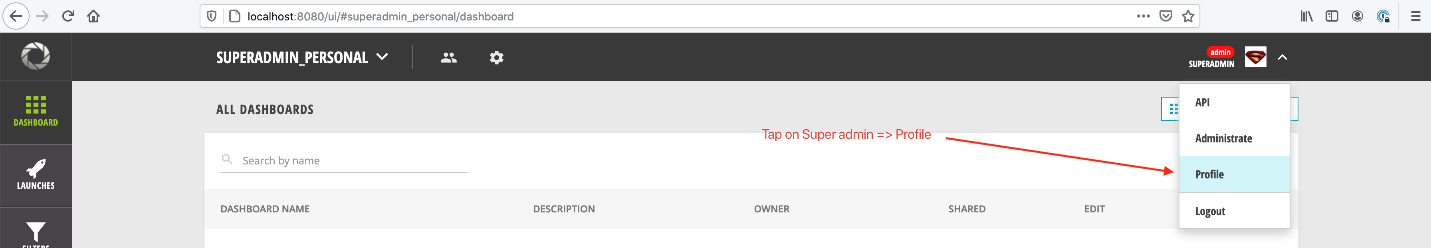
Tap on user profile settings



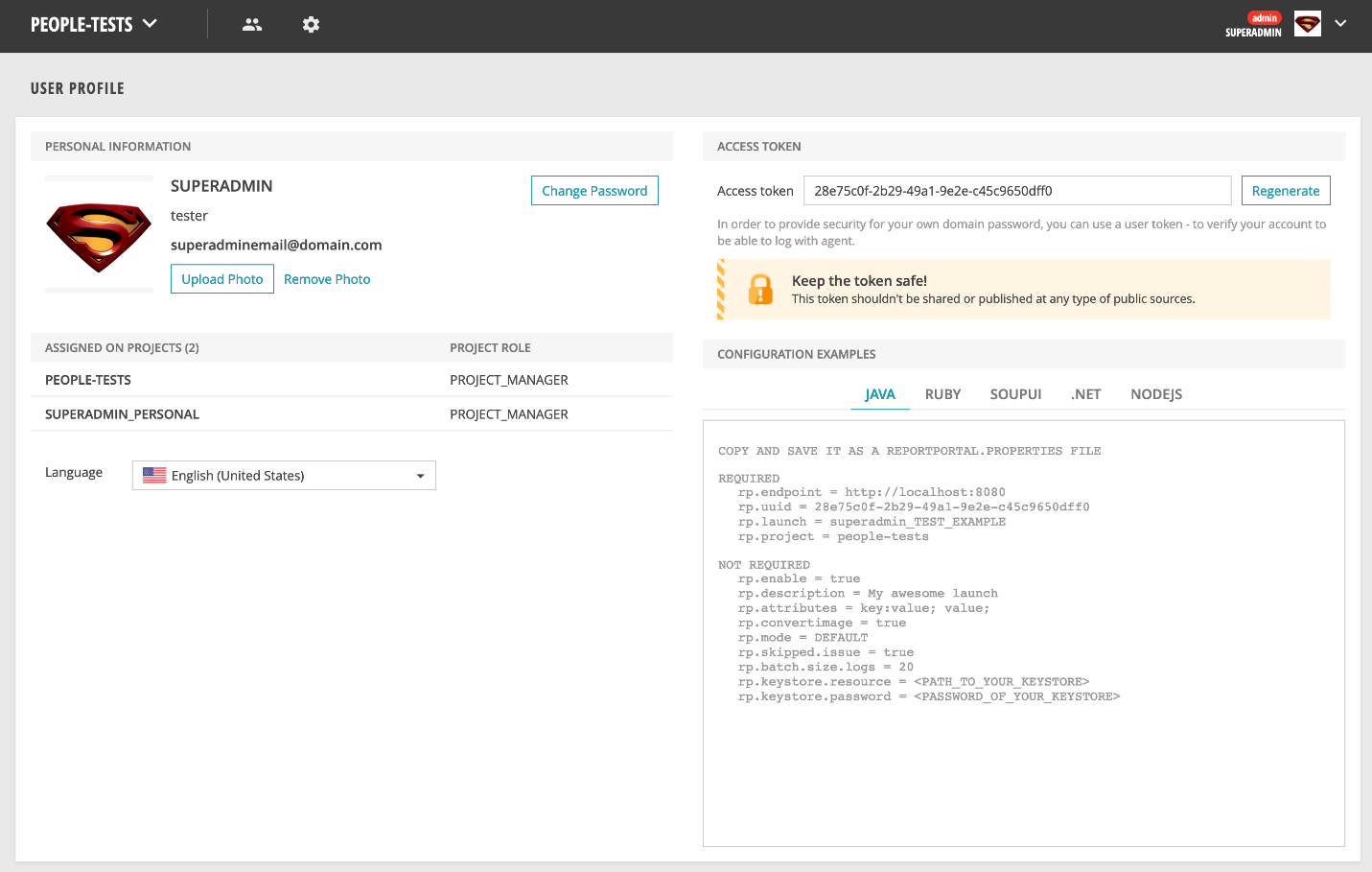
Tap on create a new project and enter a name, we’ll name our project people-tests



Open Profile



You can access the project settings which would be updated in our framework project.



Copy the required fields into a pytest.ini file in the project root

The pytest.ini file allows you to specify some default values that are auto loaded by pytest. You can also pass these manually via the command line but it can become quite tedious quite fast to pass these on every execution.

At a minimum, ensure you have the below properties present in the pytest.ini file

rp\_uuid = 28e75c0f-2b29-49a1-9e2e-c45c9650dff0 (unique id of the project)

rp\_endpoint = http://localhost:8080 (where report portal is hosted)

rp\_project = people-tests (name of the project)

rp\_launch = people-tests (what the launch will be called)

Below is a complete sample file

[pytest]

rp\_uuid = 28e75c0f-2b29-49a1-9e2e-c45c9650dff0

rp\_endpoint = http://localhost:8080

rp\_project = people-tests

rp\_launch = people-tests

rp\_launch\_attributes = 'PyTest' 'Smoke'

rp\_launch\_description = 'Smoke test'

rp\_ignore\_errors = True

rp\_ignore\_attributes = 'xfail' 'usefixture'

### Step 2: Add session-scoped logger

Pytest agent needs the plugin to be installed and registered.

Execute below:

pipenv install pytest-reportportal

We will add a **logger** method in the module scoped **conftest.py** file, which would be used in all the test methods to send different types of logs to report portal’s elasticsearch database.

tests/conftest.py

import pytest

from pytest\_reportportal import RPLogger, RPLogHandler

@pytest.fixture(scope="session")

def logger(request):

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(logging.DEBUG)

# Create a handler for Report Portal if the service has been

# configured and started.

if hasattr(request.node.config, 'py\_test\_service'):

# Import Report Portal logger and handler to the test module.

logging.setLoggerClass(RPLogger)

rp\_handler = RPLogHandler(request.node.config.py\_test\_service)

# Add additional handlers if it is necessary

console\_handler = logging.StreamHandler(sys.stdout)

console\_handler.setLevel(logging.INFO)

logger.addHandler(console\_handler)

else:

rp\_handler = logging.StreamHandler(sys.stdout)

# Set INFO level for Report Portal handler.

rp\_handler.setLevel(logging.INFO)

return logger

Here we make use of the standard python logger class and set RPLogger as the LoggerClass, along with even a console handler if needed. Notice that we’ve marked this fixture as session scope, This ensures the logger is reused and keeps on recording results to the appropriate project in the report portal throughout one test suites run

### Step 3: Add logger to tests

We’ll add the newly created logger as a dependency into the test method and pytest would auto-resolve this for us.

def test\_read\_all\_has\_kent(logger):

"""

Test on hitting People GET API, we get a user named kent in the list of people

"""

response = client.read\_all\_persons()

assert\_that(response.status\_code).is\_equal\_to(requests.codes.ok)

logger.info("User successfully read")

assert\_people\_have\_person\_with\_first\_name(response, first\_name='Kent')

Notice, we’ve added a docstring to the test method

"""

Test on hitting People GET API, we get a user named kent in the list of people

"""

Report portal pytest plugin updates all method docstrings as an actual test case description

Also, once the status code check passes, we can log the message via the below line:

logger.info("User successfully read")

### Step 4: Execute tests

Execute the below command to push logs to report portal. Notice that you would need to add –reportportal flag

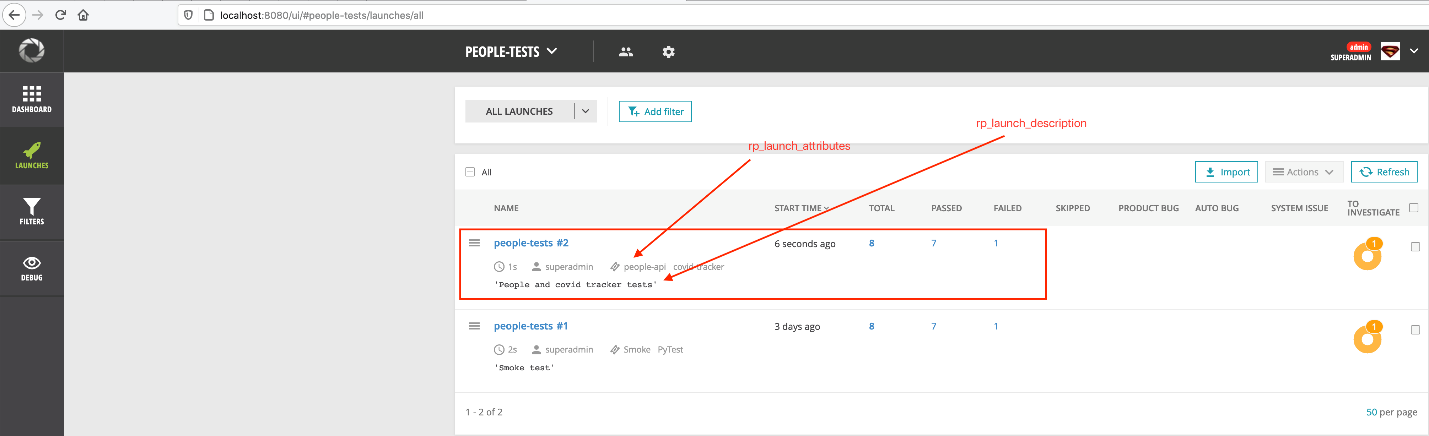
python -m pytest ./tests --reportportal

Don’t forget to ensure the people-api and covid-tracker services are up, Refer to Post 2 and Post 4, In case you need a refresher on how to start these on the local machine.

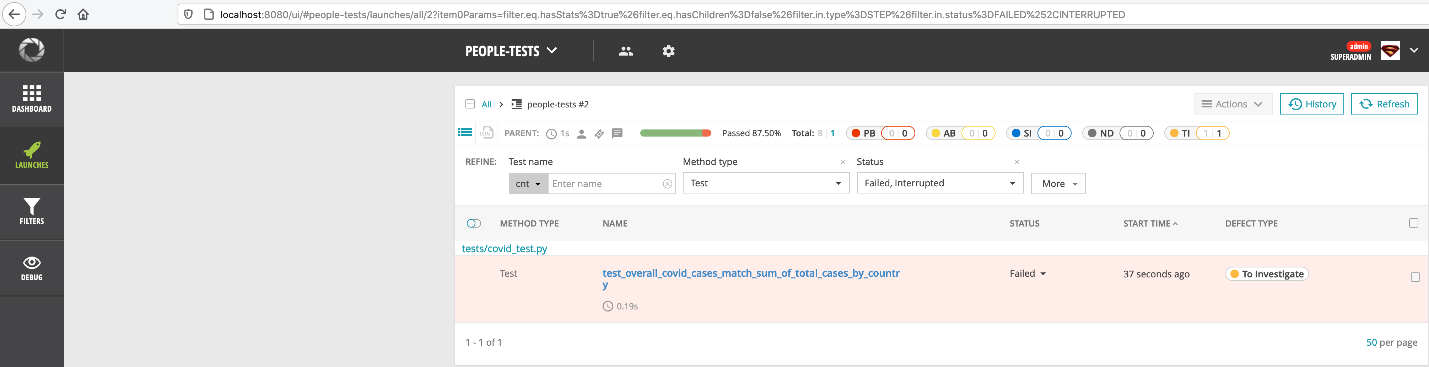
### Step 5: Analysis in report portal

Once run, let’s see how the results would look like in report portal

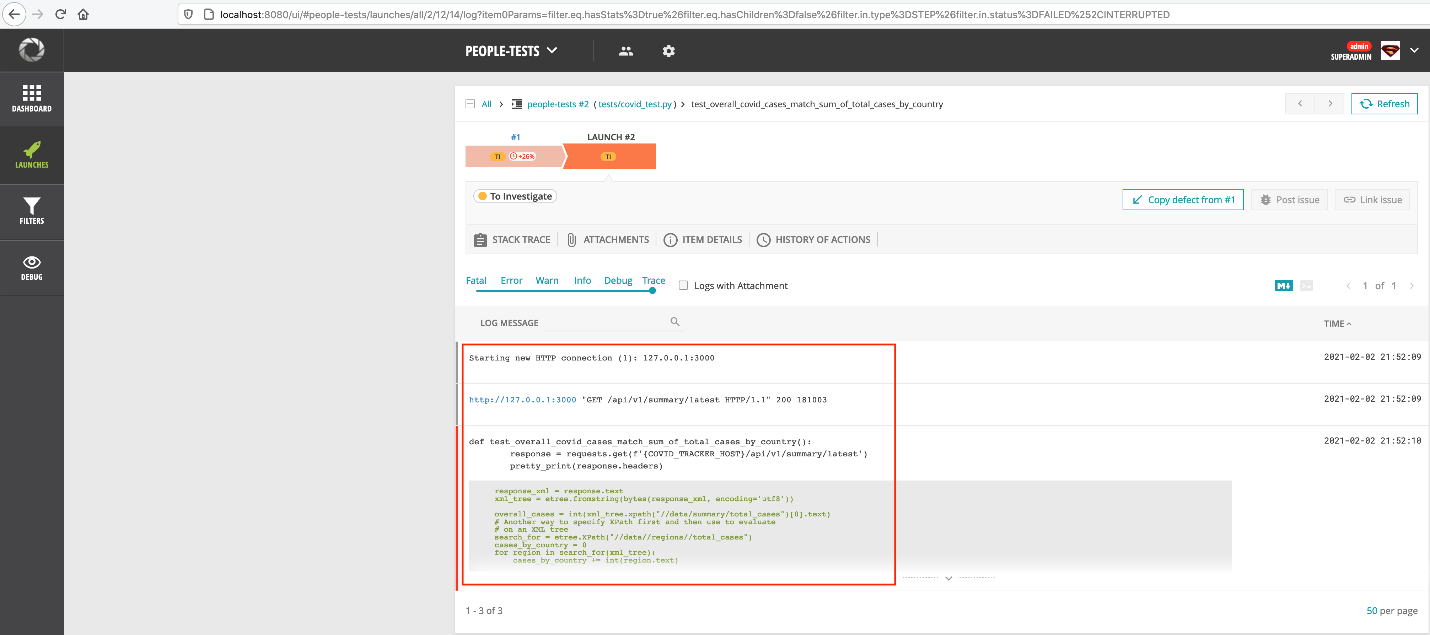
Below you can see the overall results for the test run. Notice the rp\_launch\_attributes and rp\_launch\_description are displayed on the UI.



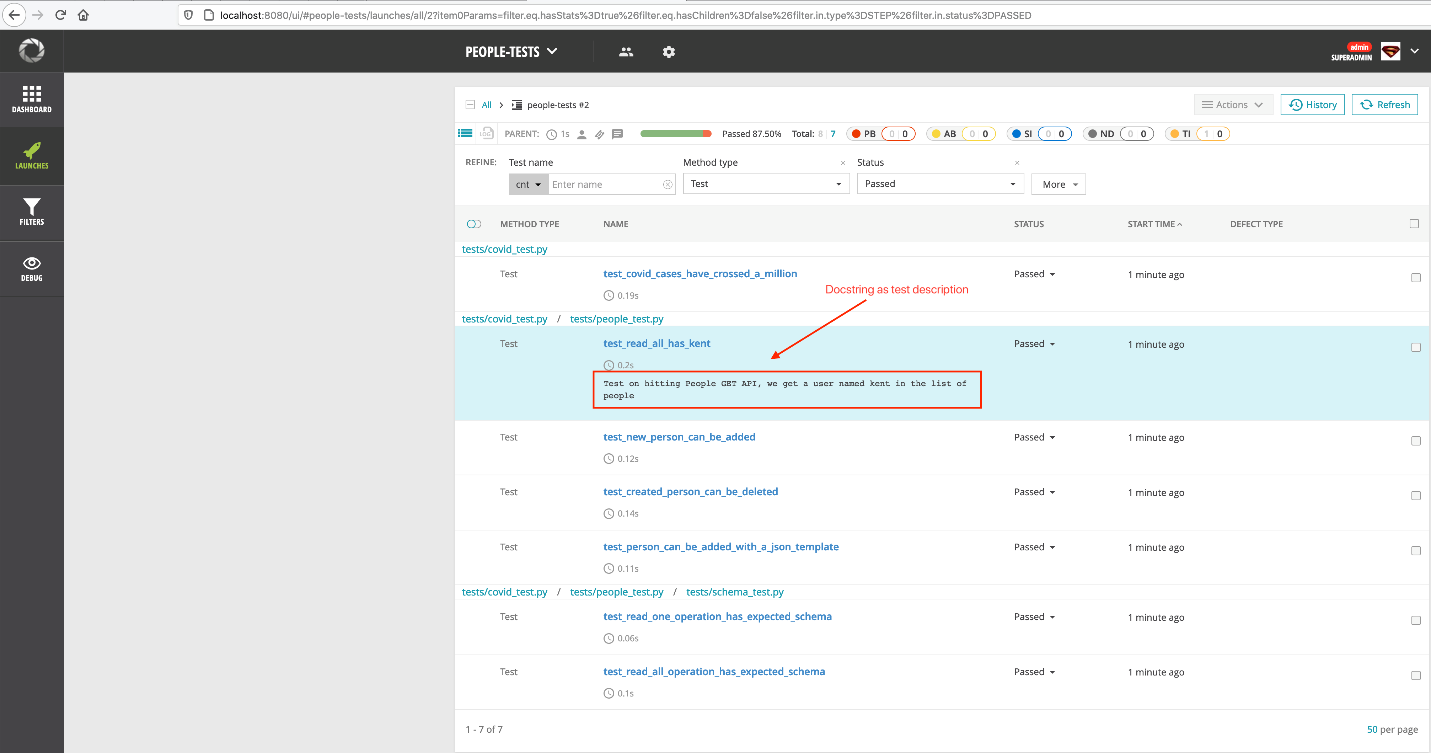
You can select Failed to see the test failure details



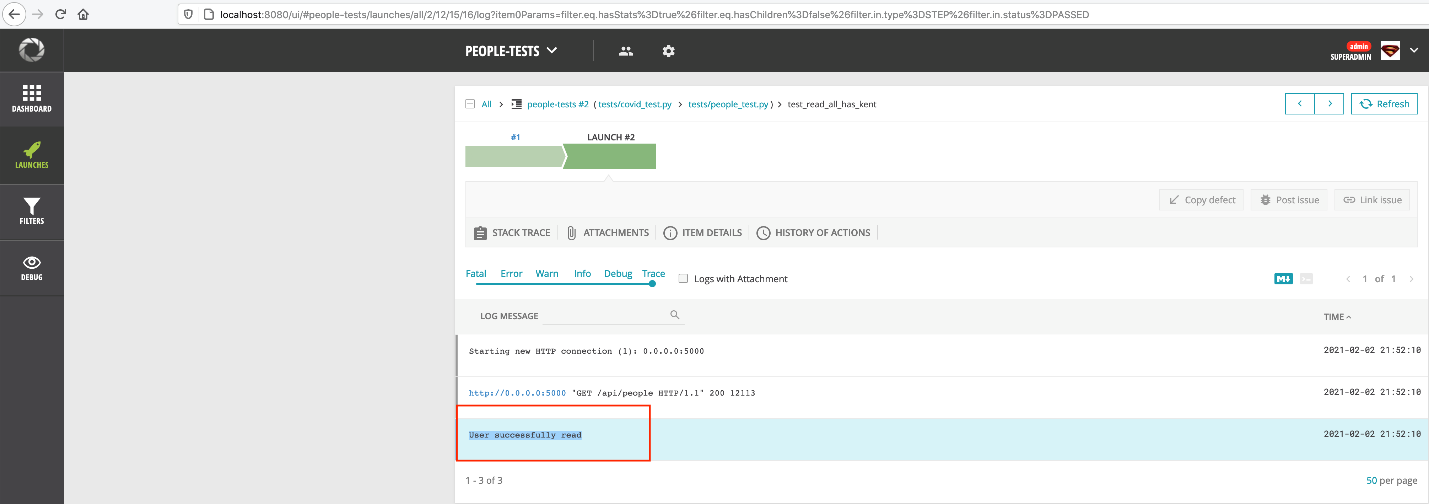
You can also see a detailed stack trace for any failures that might have occurred



Also, the docstring is displayed as the test method description



And finally, if you open the passing test test\_read\_all\_has\_kent, you can see that our log message is printed



## Conclusion

While the ReportPortal might not be the perfect reporting solution out there, it does take care of a lot of common use cases when it comes to Test reporting and easily integrates with your test framework of choice.

I hope this post helps you understand how to set this up for your own python-based framework. Do check out GitHub readme for [agent-python-pytest](https://github.com/reportportal/agent-python-pytest) to understand some more features that you can make use of

# [Python API test automation framework (Part 9) Running tests in parallel](https://automationhacks.io/2021/02/23/python-api-automation-framework-part9-running-tests-in-parallel)

This one is gonna be short and sweet 😉

As the size of your automated test suites grows from 10 to 100 to 1000 automated cases, the only way to scale your automated tests is to run them in parallel.

Often Test automation engineers start thinking about parallel runs too late in the game at which point modifying your automation is a very tedious task. Patterns that work for serial execution sometimes can be a blocker for parallel runs.

If you are completely new to this problem space and terms like **concurrency, threads, processes and parallel testing** sound like greek, don’t worry 😉, no one knows about it till they really encounter it and solve it for themselves. I promise to write a more detailed blog around this topic in the future. Keep watching this space out for it.

I recommend going through These amazing chapter in [MIT Open courseware 6.031 course on Software Construction](http://web.mit.edu/6.031/www/fa17/) on:

* [Concurrency](http://web.mit.edu/6.031/www/fa17/classes/19-concurrency/) and
* [Thread safety](http://web.mit.edu/6.031/www/fa17/classes/20-thread-safety/)
* Or read [this](https://automationpanda.com/2018/01/21/to-infinity-and-beyond-a-guide-to-parallel-testing/) useful post by Andy knight

Done reading? 😇

Let’s proceed.

How can we set up our current framework to run our tests in parallel?

Pytest provides a wonderful plugin called [pytest-xdist](https://github.com/pytest-dev/pytest-xdist) just for this purpose that makes it quite easy to achieve this

## Setup

Install below modules via pipenv

# Install pytest-xdist plugin

pipenv install pytest-xdist

pipenv install pytest-xdist"[psutil]"

# Alternatively

# After switching to example/09\_running\_tests\_in\_parallel branch

pipenv install

## Running in parallel

Run below command to run your cases on multiple cores in your machine.

Here -n mentions the number of CPU’s to send the tests to. You can either give a fixed no like 2, 3 or auto in case you want pytest to determine the no of processes (as per no of CPU cores)

python -m pytest -n auto

Notice gw0 [8] / gw1 [8] / gw2 [8] / gw3 [8] / gw4 [8] / gw5 [8] in run logs, these indicate different worker processes that pytest-xdist uses to run your tests.

platform darwin -- Python 3.8.7, pytest-6.2.2, py-1.10.0, pluggy-0.13.1

rootdir: /Users/gauravsingh/self/course-api-framework-python, configfile: pytest.ini

plugins: reportportal-5.0.8, xdist-2.2.1, forked-1.3.0

gw0 [8] / gw1 [8] / gw2 [8] / gw3 [8] / gw4 [8] / gw5 [8]

......F.

1 failed, 7 passed in 1.63s

For a small test suite like this, the cost of multiprocessing results in longer runtime however this is easily amortized for a large enough test suite.

## Customization

You can tell pytest to run all cases within a single module or all methods in a Test class within a single process by adding flag --dist loadscope

python -m pytest -n auto ./tests --reportportal --dist loadscope

Pytest xdist plugin provides many useful features like:

* running these tests within a boxed subprocess,
* over different platforms,
* getting unique id for the test run and many more.

If you want to learn more, feel free to checkout their github readme as a starting point.

[pytest-dev/pytest-xdist: pytest plugin for distributed testing and loop-on-failures testing modes.](https://github.com/pytest-dev/pytest-xdist)

## Conclusion

This was the final chapter in this long series, I hope you had as much fun 😏 as I had while putting this up.

**------END-----**