API testing in Python

using the requests library

An open source workshop by …

Originally created by Bas Dijkstra - bas@ontestautomation.com - https://www.ontestautomation.com

What are we going to do?

_RESTful APIs

_requests

_Hands-on exercises

Preparation

_Install Python 3

_Install PyCharm (or any other IDE)

_Import project into IDE
  _https://github.com/basdijkstra/requests-workshop

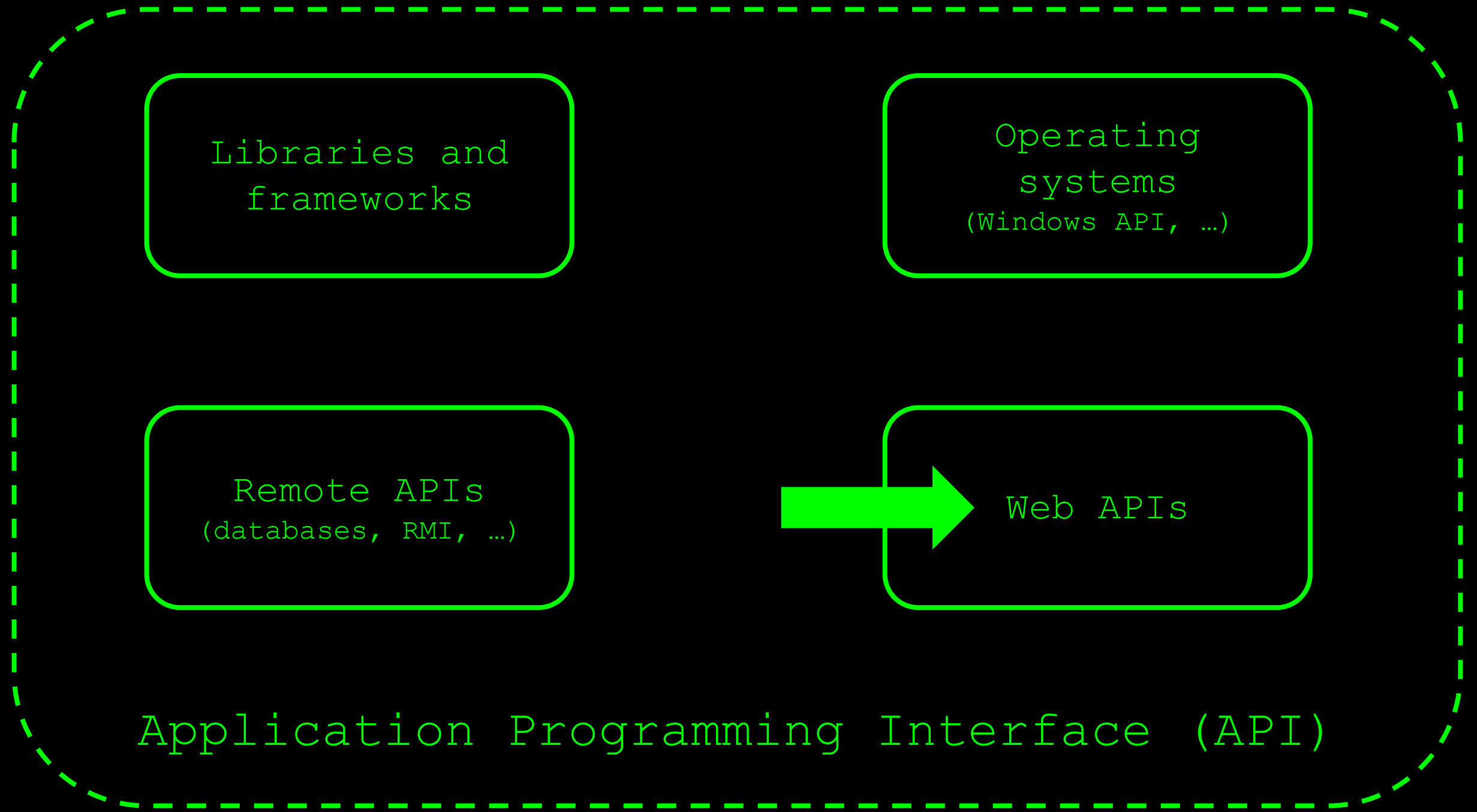
_Install dependencies, from project root:

  pip install -r requirements.txt

So, what is an API?

*"An **application programming interface (API)** is an interface or communication protocol between different parts of a computer program intended to simplify the implementation and maintenance of software"*

https://en.wikipedia.org/wiki/Application_programming_interface

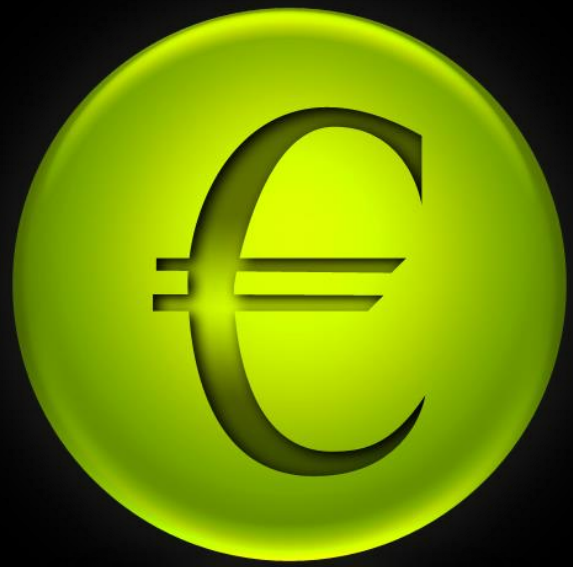

Libraries and frameworks
Operating systems
(Windows API, …)
Remote APIs
(databases, RMI, …)
Web APIs
Application Programming Interface (API)

From now on, I'll refer to these Web APIs simply as 'APIs'

Where are APIs used?

Mobile

Internet of Things



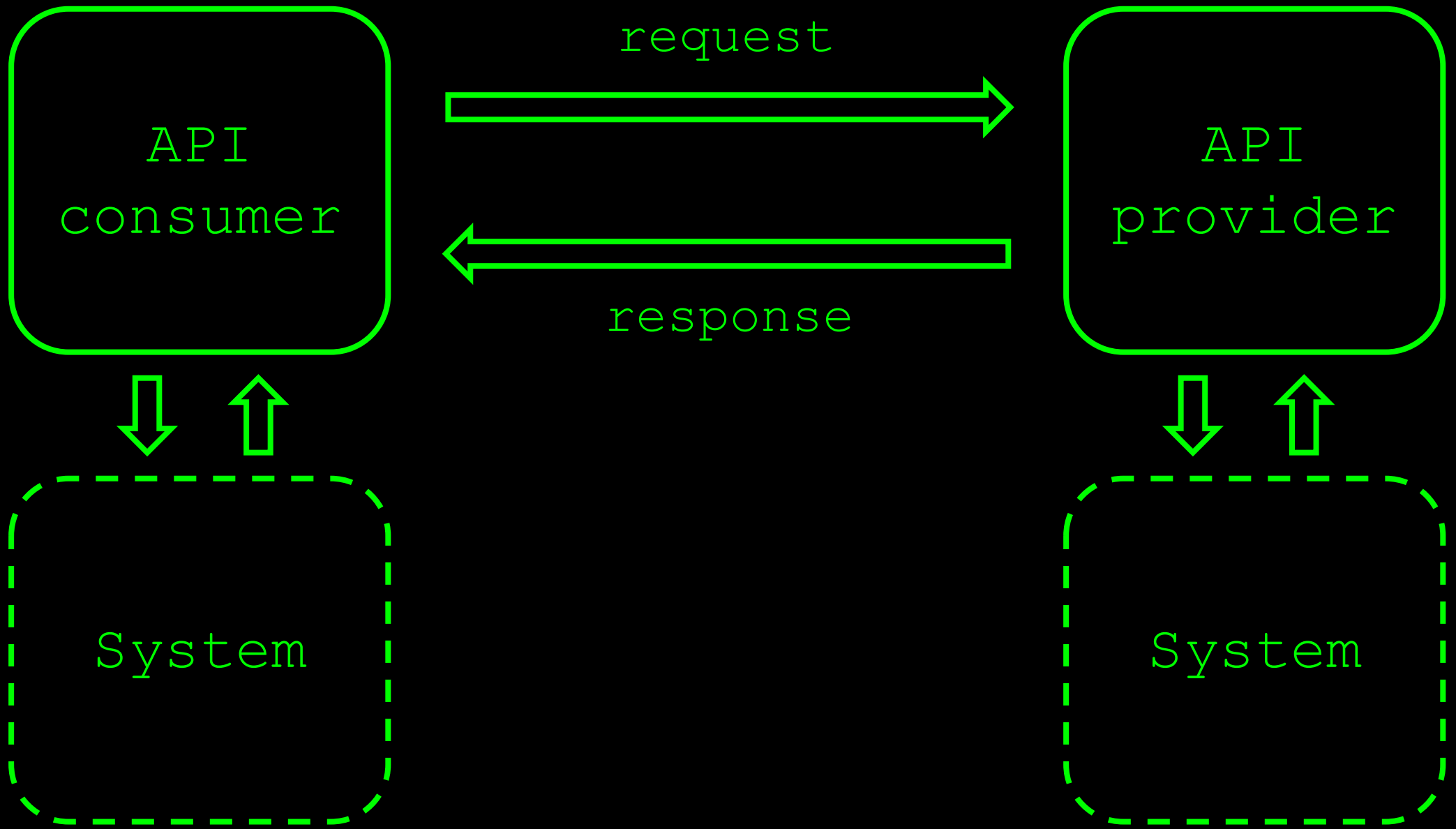
API economy

Where are APIs used?

Web applications

Microservices architectures

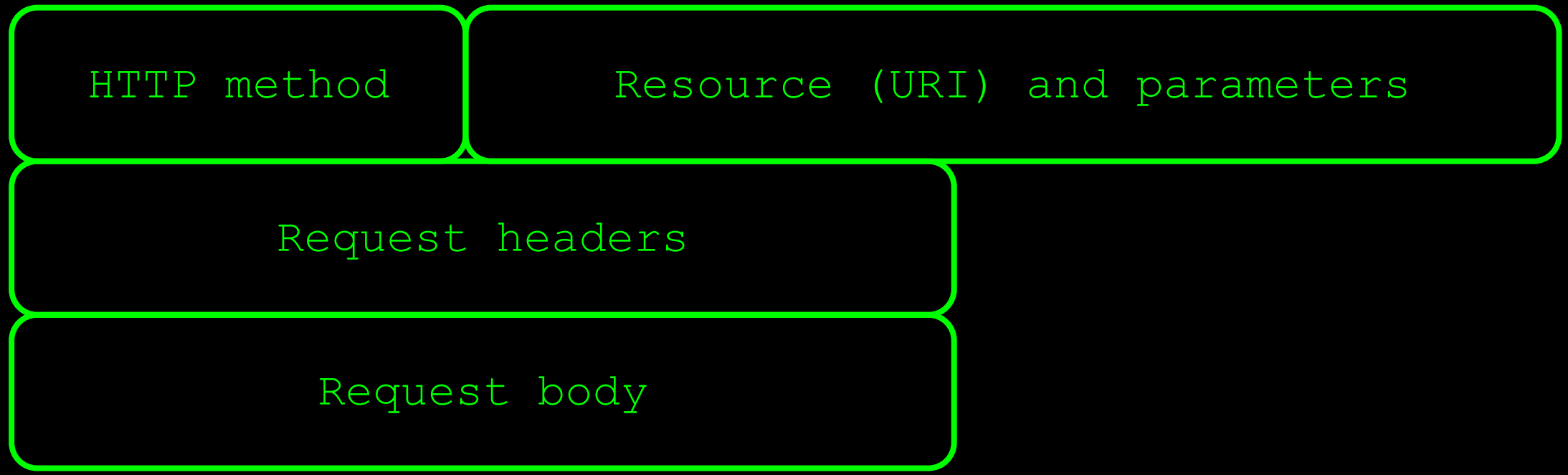
APIs are commonly used to exchange data between two parties

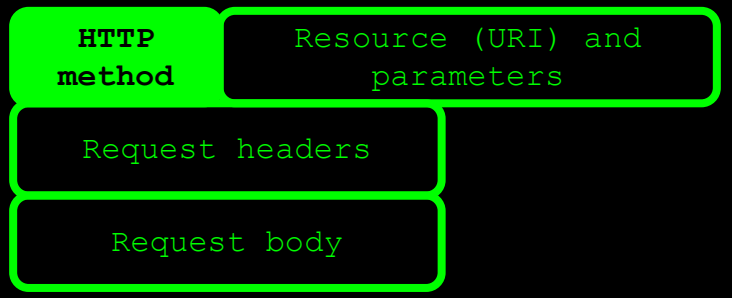

request
API
consumer
API
provider
response
System
System

SOAP and REST

	SOAP	**REST**
Protocol	HTTP, SMTP, …	HTTP
Message format	XML	XML, JSON, text, …
Specification	WSDL	WADL, RAML, Swagger, …
Standardized?	Yes	No

A REST API request





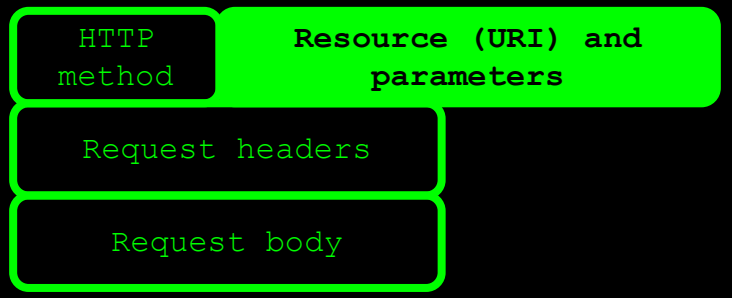
HTTP methods

_GET, POST, PUT, PATCH, DELETE, OPTIONS, …

_CRUD operations on data

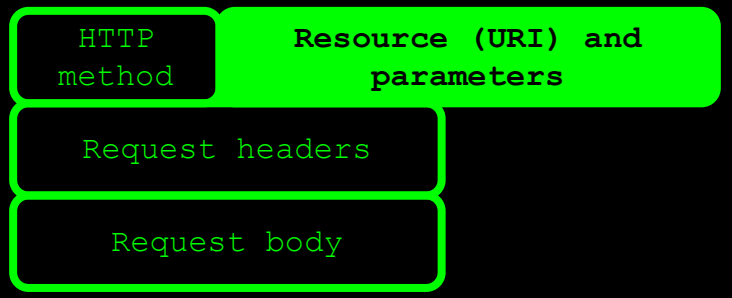
POST	Create
GET	Read
PUT / PATCH	Update
DELETE	Delete
…	…

_Conventions, not standards!



Resources and parameters

- Uniform Resource Identifier
- Uniquely identifies the resource to operate on
- Can contain parameters
  - Query parameters
  - Path parameters

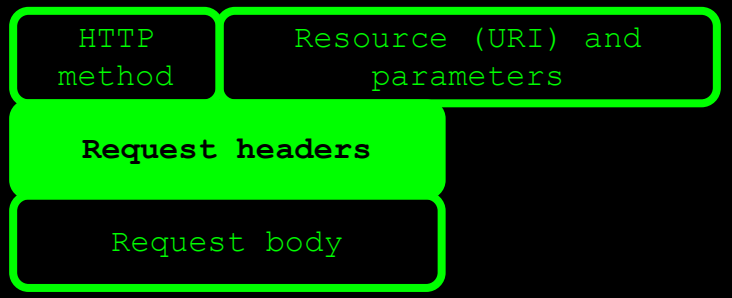


Resources and parameters

- Path parameters
  - http://api.zippopotam.us/us/90210
  - http://api.zippopotam.us/ca/B2A

- Query parameters
  - http://md5.jsontest.com/?text=testcaseOne
  - http://md5.jsontest.com/?text=testcaseTwo

- There is no official standard!

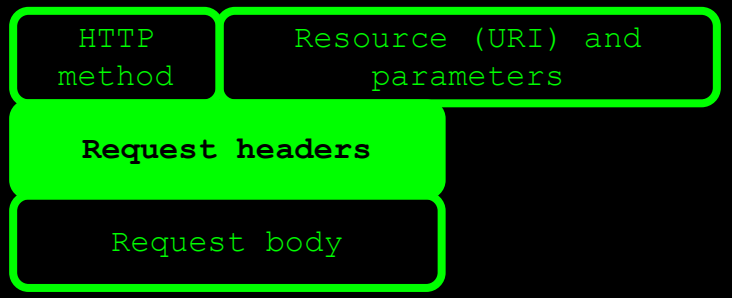


Request headers

- Key-value pairs

- Can contain metadata about the request body
  - Content-Type (what data format is the request body in?)
  - Accept (what data format would I like the response body to be in?)
  - …

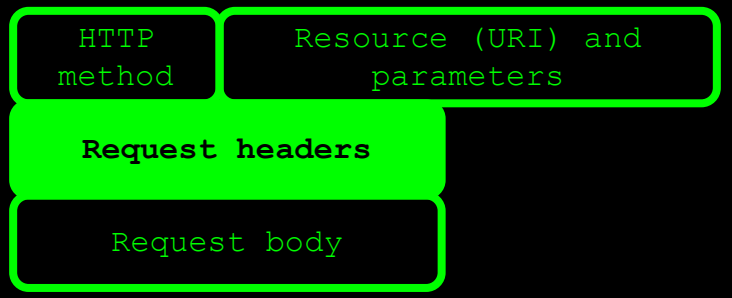
- Can contain session and authorization data
  - Cookies
  - Authorization tokens
  - …



Authorization: Basic

- Username and password sent with every request
- Base64 encoded (not really secure!)
- Ex: username = aladdin and password = opensesame

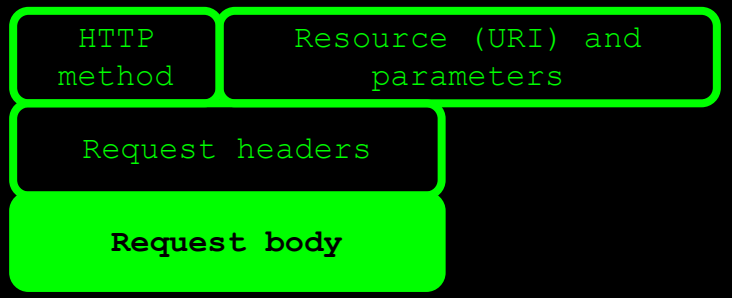
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l



Authorization: Bearer

- Token with limited validity is obtained first
- Token is then sent with all subsequent requests
- Most common mechanism is OAuth(2)
- JWT is a common token format

Authorization: Bearer RsT5OjbzRn430zqMLgV3Ia



Request body

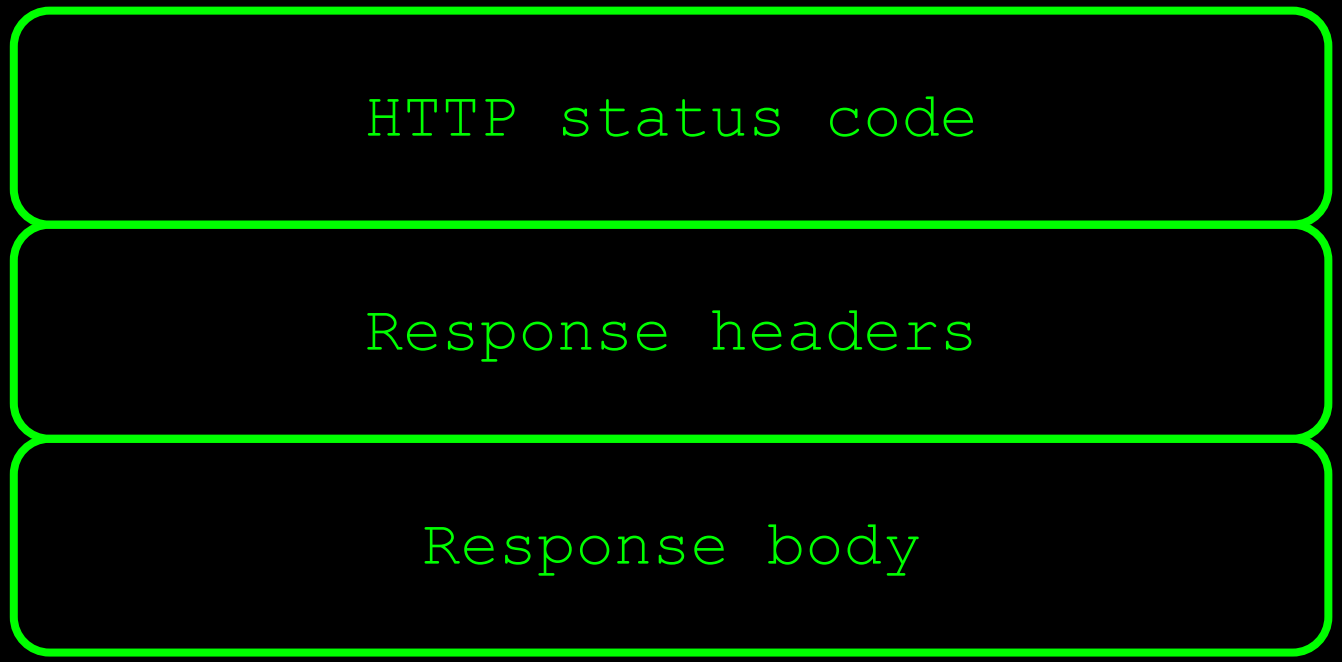
_Data to be sent to the provider

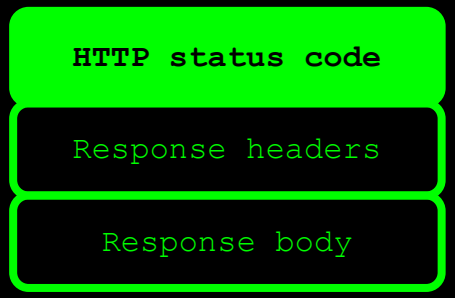
_REST does not prescribe a specific data format

_Most common:
  _JSON
  _XML
  _Plain text

_Other data formats can be sent using REST, too

A REST API response



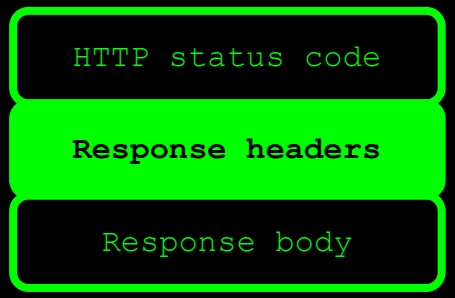


HTTP status code

_Indicates result of request processing by provider

_Five different categories

_1XX	Informational	100 Continue
_2XX	Success	200 OK
_3XX	Redirection	301 Moved Permanently
_4XX	Client errors	400 Bad Request
_5XX	Server errors	503 Service Unavailable

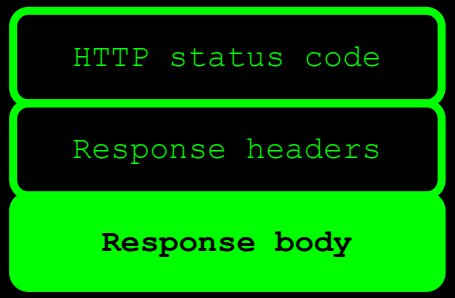


Response headers

- Key-value pairs

- Can contain metadata about the response body
  - Content-Type (what data format is the response body in?)
  - Content-Length (how many bytes in the response body?)

- Can contain provider-specific data
  - Caching-related headers
  - Information about the server type



Response body

_Data returned by the provider

_REST does not prescribe a specific data format

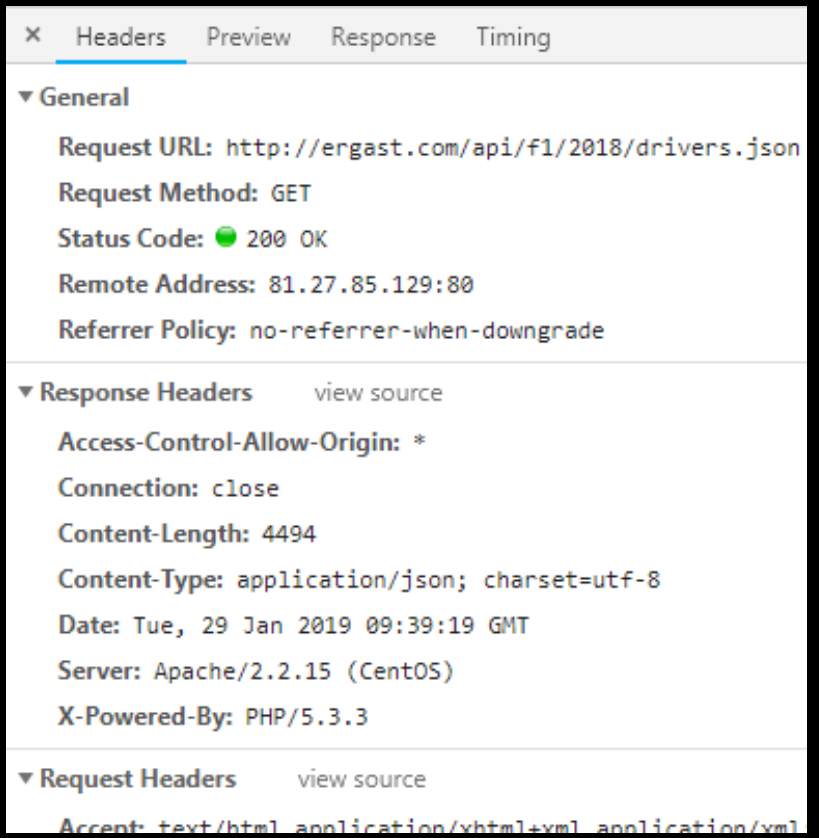
_Most common:
  _JSON
  _XML
  _Plain text

_Other data formats can be sent using REST, too

An example

_GET http://ergast.com/api/f1/2018/drivers.json


```
{
  - MRData: {
        xmlns: "http://ergast.com/mrd/1.4",
        series: "f1",
        url: "http://ergast.com/api/f1/2018/drivers.json",
        limit: "30",
        offset: "0",
        total: "20",
      - DriverTable: {
            season: "2018",
          - Drivers: [
              - {
                    driverId: "alonso",
                    permanentNumber: "14",
                    code: "ALO",
                    url: "http://en.wikipedia.org/wiki/Fernando_Alonso",
                    givenName: "Fernando",
                    familyName: "Alonso",
                    dateOfBirth: "1981-07-29",
                    nationality: "Spanish"
                },
              - {
                    driverId: "bottas",
                    permanentNumber: "77",
                    code: "BOT"
```

Why I ♥ testing at the API level

_Tests run much faster than UI-driven tests

_Tests are easier to stabilize than UI-driven tests

_Tests have a broader scope than unit tests

_Business logic is often exposed at the API level

Tools for testing RESTful web services

- Free / open source
  - Postman, SoapUI, REST Assured, requests, …
- Commercial
  - Parasoft SOAtest, SoapUI Pro, …
- Build your own (using HTTP libraries for your language of choice)

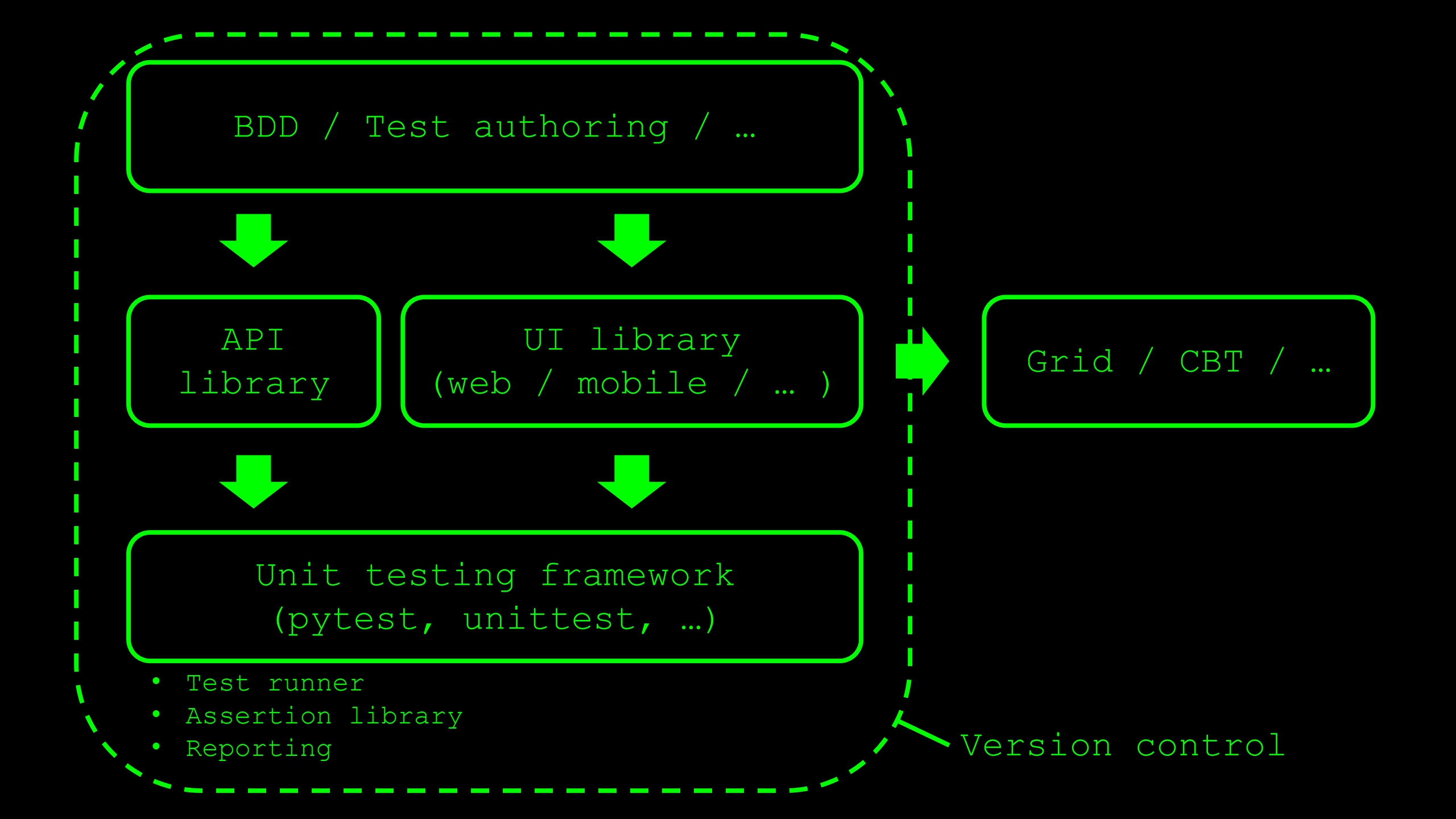
Python library for interacting with REST APIs

"Requests is an elegant and simple HTTP library for Python, built for human beings."

requests

pip install requests

https://requests.readthedocs.io/en/master/


BDD / Test authoring / …
API library
UI library (web / mobile / …)
Grid / CBT / …
Unit testing framework (pytest, unittest, …)
• Test runner
• Assertion library
• Reporting
Version control

In this workshop, we'll use requests with pytest

A few example tests

Checking response status code

This import statement is required to allow using the requests library in a Python module

```python
import requests


def test_get_user_with_id_1_check_status_code_equals_200():
    response = requests.get("https://jsonplaceholder.typicode.com/users/1")
    assert response.status_code == 200
```

The response object return by the *get()* has a *status_code* property (an integer), which can be compared to a previously defined expected HTTP status code value using the pytest *assert* keyword

Checking response headers

```
def test_get_user_with_id_1_check_content_type_equals_json():
    response = requests.get("https://jsonplaceholder.typicode.com/users/1")
    assert response.headers['Content-Type'] == "application/json; charset=utf-8"
```

The response object return by the *get()* has a *headers* property (a dictionary) containing all the response headers.

As with every dictionary, Python will raise a *KeyError* when you try to access a key (i.e., a header name) that does not exist in the dictionary.

Checking a JSON body element

```python
def test_get_user_with_id_1_check_name_equals_leanne_graham():
    response = requests.get("https://jsonplaceholder.typicode.com/users/1")
    response_body = response.json()
    assert response_body["name"] == "Leanne Graham"
```

response.json() converts the JSON response body into an ordinary Python dictionary, which you can then query for specific elements

For example, *response_body['name']* refers to the top-level element name in the response body

The value of that element is equal to 'Leanne Graham' in this example, so we would expect this test to pass (and it does)


```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

Checking nested body elements

```python
def test_get_user_with_id_1_check_company_name_equals_romaguera_crona():
    response = requests.get("https://jsonplaceholder.typicode.com/users/1")
    response_body = response.json()
    assert response_body["company"]["name"] == "Romaguera-Crona"
```

In this example, the value of *response_body['company']* is another dictionary, meaning you can access its elements using the same notation once more.

I.e., *response_body['company']['name']* points to 'Romaguera-Crona'


```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

Checking the size of an array

```
def test_get_all_users_check_number_of_users_equals_10():
    response = requests.get("https://jsonplaceholder.typicode.com/users")
    response_body = response.json()
    assert len(response_body) == 10
```

If the top-level element in the JSON response is an array, you can use the Python len() function to assert on the number of items in it

Of course, this works for all elements in a response. If a top-level JSON response body element *places* would have an array as its value, you could assert on its length using *len(response_body['places'])*

Our API under test

_Zippopotam.us

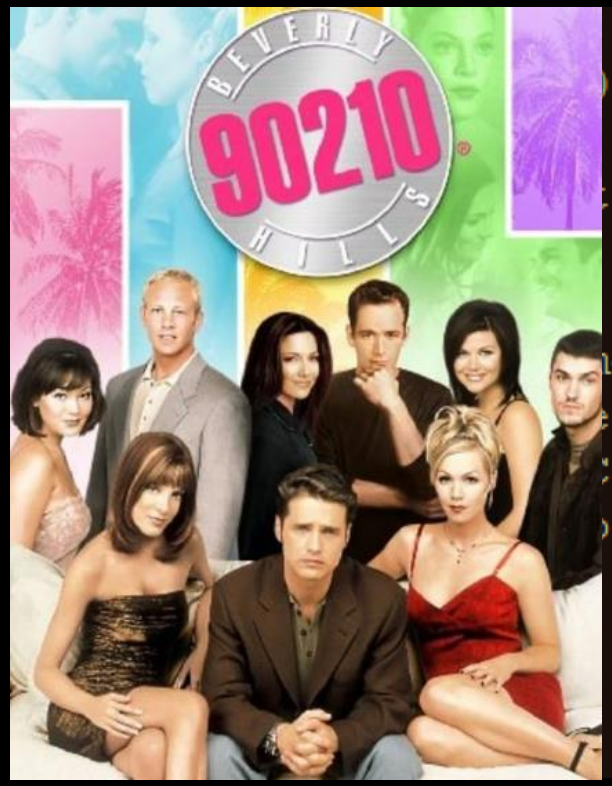
_Returns location data based on country and zip code

_http://api.zippopotam.us/

_RESTful API

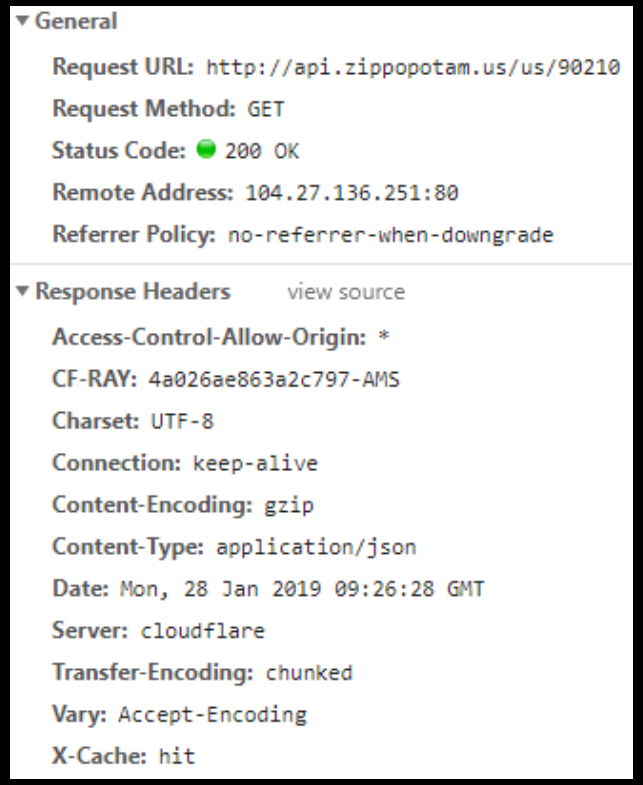
An example

_GET http://api.zippopotam.us/us/90210




```
",
States",
ion: "US",

e: "Beverly Hills",
: "-118.4065",
alifornia",
reviation: "CA",
"34.0901"
```

Now it's your turn!

_ exercises > exercises_01.py

_ run your answers (from the project root) using

pytest exercises\exercises_01.py

_ examples are in examples > examples_01.py

_ answers are in answers > answers_01.py

Exchange data between consumer and provider

GET to retrieve data from provider, POST to send data to provider, …

APIs are all about data

Business logic and calculations often exposed through APIs

Run the same test more than once…

… for different combinations of input and expected output values

Data driven testing

More efficient to do this at the API level…

… as compared to doing this at the UI level

Parameters in RESTful APIs

_Path parameters
  _http://api.zippopotam.us/us/90210
  _http://api.zippopotam.us/ca/B2A

_Query parameters
  _http://md5.jsontest.com/?text=testcaseOne
  _http://md5.jsontest.com/?text=testcaseTwo

_There is no official standard!

Data driven API testing

```
test_data_users = [
    (1, "Leanne Graham"),
    (2, "Ervin Howell"),
    (3, "Clementine Bauch")
]

@pytest.mark.parametrize("userid, expected_name", test_data_users)
def test_get_data_for_user_check_name(userid, expected_name):
    response = requests.get(f"https://jsonplaceholder.typicode.com/users/{userid}")
    response_body = response.json()
    assert response_body["name"] == expected_name
```

First, create a list of test data tuples…

… then pass the tuple as an argument to the @pytest.mark.parametrize marker…

… then feed the test data values as arguments to the test method…

… and use them in the test as required.

```
collected 3 items

examples_02.py ...                                                    [100%]

============================== 3 passed in 0.49s ===============================
```

The test will run once for each test data tuple in the list

Working with external data sources

Reading a .csv file

```
import csv
```

Enables using the builtin *csv* library in Python

```
1,Leanne Graham
2,Ervin Howell
3,Clementine Bauch
```

This is what our actual .csv file contains

The *with* statement is a context manager (used to allocate and release resources effectively)

```
def read_data_from_csv():
    test_data_users_from_csv = []
    with open("examples/test_data_users.csv", newline='') as csvfile:
        data = csv.reader(csvfile, delimiter=',')
        for row in data:
            test_data_users_from_csv.append(row)
    return test_data_users_from_csv
```

Read all the data in the .csv, using the comma as a field delimiter

Go through all rows in the .csv file, one by one

Add each row as a tuple to the test data list, creating the same data structure as we've seen in the previous example

Using .csv data to drive tests

```
def read_data_from_csv():
    test_data_users_from_csv = []
    with open("examples/test_data_users.csv", newline='') as csvfile:
        data = csv.reader(csvfile, delimiter=',')
        for row in data:
            test_data_users_from_csv.append(row)
    return test_data_users_from_csv
```

```
@pytest.mark.parametrize("userid, expected_name", read_data_from_csv())
def test_get_location_data_check_place_name_with_data_from_csv(userid, expected_name):
    response = requests.get(f"https://jsonplaceholder.typicode.com/users/{userid}")
    response_body = response.json()
    assert response_body["name"] == expected_name
```

Instead of feeding the pytest marker a list directly, we can also feed it the return value of a method, as long as that method returns test data in the required structure (i.e., a list of test data tuples)

Now it's your turn!

_ exercises > exercises_02.py

_ run your answers from the project root using

pytest exercises\exercises_02.py

_ examples are in examples > examples_02.py

_ answers are in answers > answers_02.py

Creating a JSON request body

```python
import random


def create_new_post_object():
    return {
        "name": "John Smith",
        "address": {
            "street": "Main Street",
            "number": random.randint(1000, 9999),
            "zipCode": 90210,
            "city": "Beverly Hills"
        }
    }
```

Write a function that returns a dictionary

Create and return a dictionary containing the data to be converted to and POSTed as JSON

You can even generate and use random or other types of dynamic values

POSTing a JSON request body

```
import random


def create_new_post_object():

    return {
        "name": "John Smith"
    }
```

Use the data dictionary as the value of the *json* argument to have requests convert it to JSON before POSTing it

```
def test_send_json_with_unique_number_check_status_code():
    response = requests.post("https://postman-echo.com/post", json=create_new_post_object())
    print(response.request.body)
    assert response.status_code == 200
```

```
C:\Git\requests-workshop>pytest -s examples\examples_03.py
```

This disables output capturing by pytest, so all print() statements will be sent to the stdout / console

```
{"name": "John Smith", "address": {"street": "Main Street", "number": 5248, "zipCode": 90210, "city": "Beverly Hills"}}
```

Now it's your turn!

_ exercises > exercises_03.py

_ run your answers from the project root using

pytest exercises\exercises_03.py

_ examples are in examples > examples_03.py

_ you will need to Google some things yourself

_ answers are in answers > answers_03.py

Create XML request body using a docstring

```python
def use_xml_string_block():
    return '''
    <users>
        <user>
            <id>5b4832b4-da4c-48b2-8512-68fb49b6
            <name>John Smith</name>
            <phone type="mobile">0612345678</phone>
            <phone type="landline">0992345678</phone>
        </user>
    </users>
    '''
```

Triple quotes can be used in Python to create multiline strings

```xml
<users>
    <user>
        <id>5b4832b4-da4c-48b2-8512-68fb49b69de1</id>
        <name>John Smith</name>
        <phone type="mobile">0612345678</phone>
        <phone type="landline">0992345678</phone>
    </user>
</users>
```

Pass the string as a value to the *data* argument to POST it raw (without processing)

```python
def test_send_xml_using_xml_string_block():
    response = requests.post("http://httpbin.org/anything", data=use_xml_string_block())
    print(response.request.body)
    assert response.status_code == 200
```

Create XML request body using lxml

```python
from lxml import etree


def create_xml_object():
    users = etree.Element("users")
    user = etree.SubElement(users, "user")
    user_id = etree.SubElement(user, "id")
    user_id.text = unique_number
    name = etree.SubElement(user, "name")
    name.text = "John Smith"
    phone1 = etree.SubElement(user, "phone")
    phone1.set("type", "mobile")
    phone1.text = "0612345678"
    phone2 = etree.SubElement(user, "phone")
    phone2.set("type", "landline")
    phone2.text = "0992345678"

    return users
```

Import the etree library from lxml to create XML payloads programmatically

Create an XML root element…

Add child elements…

Set element values…

```xml
<users>
    <user>
        <id>5b4832b4-da4c-48b2-8512-68fb49b69de1</id>
        <name>John Smith</name>
        <phone type="mobile">0612345678</phone>
        <phone type="landline">0992345678</phone>
    </user>
</users>
```

Add attributes and their values

Send XML created using lxml

```python
from lxml import etree


def create_xml_object():
    users = etree.Element("users")
    user = etree.SubElement(users, "user")
    user_id = etree.SubElement(user, "id")
    user_id.text = unique_number
    name = etree.SubElement(user, "name")
    name.text = "John Smith"
    phone1 = etree.SubElement(user, "phone")
    phone1.set("type", "mobile")
    phone1.text = "0612345678"
    phone2 = 
    phone2.se
    phone2.te

    return us
```

```xml
<users>
    <user>
        <id>5b4832b4-da4c-48b2-8512-68fb49b69de1</id>
        <name>John Smith</name>
        <phone type="mobile">0612345678</phone>
        <phone type="landline">0992345678</phone>
    </user>
</users>
```

The *tostring()* method returns a string representation of the ElementTree, which we then pass to the *data* argument to send it as a raw string request body

```python
def test_send_xml_using_lxml_etree():
    xml = create_xml_object()
    response = requests.post("http://httpbin.org/anything", data=etree.tostring(xml))
    print(response.request.body)
    assert response.status_code == 200
```

Now it's your turn!

_ exercises > exercises_04.py

_ run your answers from the project root using

pytest exercises\exercises_04.py

_ examples are in examples > examples_04.py

_ answers are in answers > answers_04.py

Checking response XML root element

```python
def test_check_root_of_xml_response():
    response = requests.get(
        "https://parabank.parasoft.com/parabank/services/bank/customers/12212"
    )
    xml_response_element = etree.fromstring(response.content)
    xml_response_tree = etree.ElementTree(xml_response_element)
    root = xml_response_tree.getroot()
    assert root.tag == "customer"
    assert root.text is None
```

Parse the raw response body as an XML Element…

… convert it to an ElementTree…

… get the root element from the tree…

… check the root element name, and…

… check that it does not have a text value

```
▼<customer>
  <id>12212</id>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
 ▼<address>
    <street>1431 Main St</street>
    <city>Beverly Hills</city>
    <state>CA</state>
    <zipCode>90210</zipCode>
  </address>
  <phoneNumber>310-447-4121</phoneNumber>
  <ssn>622-11-9999</ssn>
</customer>
```

Checking response XML – find an element using *find()*

```
def test_check_specific_element_of_xml_response():
    response = requests.get(
        "https://parabank.parasoft.com/parabank/services/bank/customers/12212"
    )
    xml_response_element = etree.fromstring(response.content)
    xml_response_tree = etree.ElementTree(xml_response_element)
    first_name = xml_response_tree.find("firstName")
    assert first_name.text == "John"
    assert len(first_name.attrib) == 0
```

Extract and create the ElementTree as before…

… find the first occurrence of the *firstName* element in the tree…

… and assert on the element text and the number of attributes the element has

```
▼<customer>
  <id>12212</id>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
 ▼<address>
    <street>1431 Main St</street>
    <city>Beverly Hills</city>
    <state>CA</state>
    <zipCode>90210</zipCode>
  </address>
  <phoneNumber>310-447-4121</phoneNumber>
  <ssn>622-11-9999</ssn>
</customer>
```

Checking response XML – find all elements using *findall()*

```python
# https://docs.python.org/3/library/xml.etree.elementtree.html#elementtree-xpath
def test_use_xpath_for_more_sophisticated_checks():
    response = requests.get(
        "https://parabank.parasoft.com/parabank/services/bank/customers/12212"
    )
    xml_response_element = etree.fromstring(response.content)
    xml_response_tree = etree.ElementTree(xml_response_element)
    address_children = xml_response_tree.findall(".//address/*")
    assert len(address_children) == 4
```

```xml
<customer>
  <id>12212</id>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <address>
    <street>1431 Main St</street>
    <city>Beverly Hills</city>
    <state>CA</state>
    <zipCode>90210</zipCode>
  </address>
  <phoneNumber>310-447-4121</phoneNumber>
  <ssn>622-11-9999</ssn>
</customer>
```

Extract and create the ElementTree as before…

… find all occurrences that match the XPath expression (i.e., child elements of *address*)…

… and assert that there are four of them

Now it's your turn!

_ exercises > exercises_05.py

_ run your answers from the project root using

pytest exercises\exercises_05.py

_ examples are in examples > examples_05.py

_ you will need to Google some things yourself

_ answers are in answers > answers_05.py

The problem with ‘traditional’ REST APIs

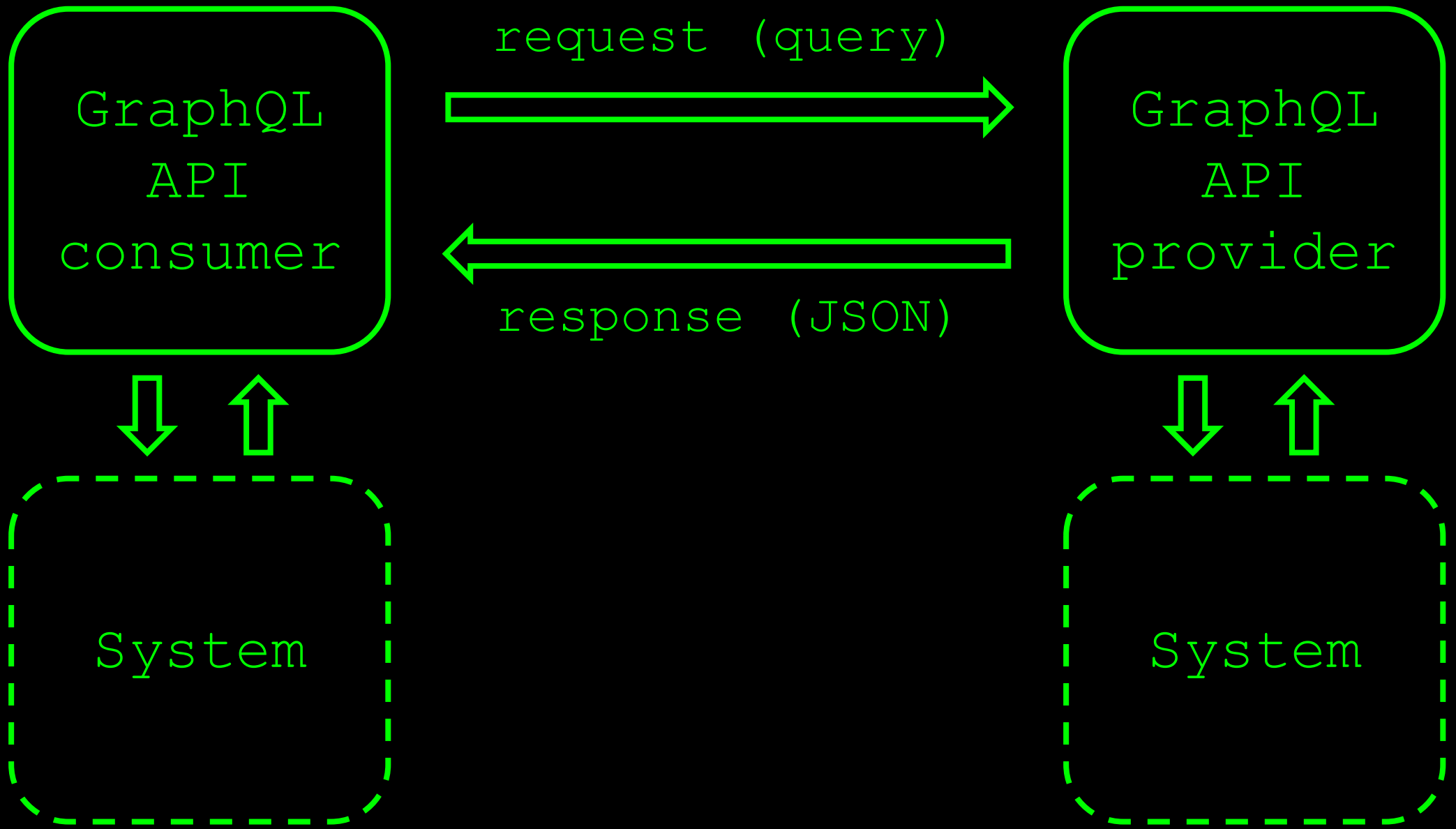
Query language for APIs…

… as well as a runtime to fulfill them

GraphQL

"Ask for what you need,
and get exactly that"

https://graphql.org


GraphQL
API
consumer
request (query)
response (JSON)
GraphQL
API
provider
System
System

Create a valid GraphQL query…

… and send it in the request body (*query*)

Sending a GraphQL query

"Ask for what you need,
and get exactly that"

These are ‘regular’ REST responses, with…

… an HTTP status code, …

GraphQL API responses

… response headers…

… and a JSON response body
containing the requested data

Sending a basic GraphQL query

```
query_weather_in_amsterdam = """
{
  getCityByName(name: "Amsterdam") {
    weather {
      summary {
        title
      }
    }
  }
}
"""
```

The query can be a simple (multiline) String

```
def test_get_weather_for_amsterdam_should_be_clear():
    response = requests.post(
        "https://graphql-weather-api.herokuapp.com/",
        json={'query': query_weather_in_amsterdam}
    )
    assert response.status_code == 200
    response_body = response.json()
    assert response_body['data']['getCityByName']['weather']['summary']['title'] == 'Clear'
```

Create the GraphQL query payload and POST it as JSON to the GraphQL endpoint, as you would do with a regular REST endpoint

The response body is regular JSON, so we know how to handle that already

Sending a parameterized GraphQL query

```
query_weather_parameterized = """
query getWeather($city: String!)
{
  getCityByName(name: $city) {
    weather {
      summary {
        title
      }
    }
  }
}
"""
```

This query allows us to specify the city name as a parameter, so we can request the weather in different cities

Define our test data rows ('test cases')

```
test_data_weather = [
    ('Amsterdam', 'Clear'),
    ('Berlin', 'Clear'),
    ('Sydney', 'Rain')
]
```

Feed the data to our test

```
@pytest.mark.parametrize('city_name, expected_weather', test_data_weather)
def test_get_weather_for_city_should_be_as_expected(city_name, expected_weather):
    response = requests.post(
        "https://graphql-weather-api.herokuapp.com/",
        json={'query': query_weather_parameterized,
              'variables': {
                  'city': city_name
              }}
    )
    assert response.status_code == 200
    response_body = response.json()
    assert response_body['data']['getCityByName']['weather']['summary']['title'] == expected_weather
```

Create the GraphQL query payload, including the variables, and POST it as JSON to the GraphQL endpoint

Check the actual weather against the expected weather

Now it's your turn!

_ exercises > exercises_06.py

_ run your answers from the project root using

pytest exercises\exercises_06.py

_ examples are in examples > examples_06.py

_ answers are in answers > answers_06.py