tapi Documentation

Release 0.1

Jimmy John

July 02, 2014

Contents

Tapi is a tool to automate the testing of your Application Programmer Interfaces (APIs).

CHAPTER 1

Why use TAPI?

Once you have built up your API server, comes the important task of writing tests for it (or is it the other way around :)) Testing APIs is a relatively simple, albeit time consuming task. It typically involves:

1. Make a request to an endpoint (using one of the verbs like GET, POST, PUT etc)
2. Verify the return status code/body etc

Most folks use their favourite testing tools (e.g. python unittest) and whip up tests cases that do just this. However, for something so simple, can one get away without writing any code?

That's what Tapi tries to do. You specify the APIs you want to test in a json file and also what the return codes should be. The Tapi framework takes care of making the request and checking the return code is as expected. You can also perform other verifications like headers, content of body, executing yet another API call to ensure that some content has been correctly POSTed etc.

Thus, Tapi makes testing your APIs as easy as editing a json file.

CHAPTER 2

Features

1. Test you API without writing any code (only edit a json file)
2. Test you APIs in a much more 'natural' way by specifying urls/verbs and what the output should be
3. Verify anything from response status codes, headers, body content etc
4. Also allows verification by issuing another API call to a different endpoint to ensure a prior API call worked
5. Execute arbitrary python scripts to populate request paramaters e.g. custom headers
6. Execute arbitrary python scripts to verify response from endpoint is valid
7. Tests your APIs using your own APIs

CHAPTER 3

Dependencies

Tapi has the following dependencies:

1. Python 2.7+
2. Requests
3. Python JsonPath

CHAPTER 4

Installation

Create a virtualenv and then install via pip:

```
pip install tapi
```

CHAPTER 5

Quick Start

In order to start using Tapi you have to write a tapi.json file. The simplest tapi.json file looks like:

```
{
    "tests": [
        {
            "main": {
                "request": {
                    "url": "http://api.example.com/users"
                }
            }
        }
    ]
}
```

The above means the following:

1. There is one test in this file
2. The framework will make a GET (the default) request to the endpoint api.example.com/users
3. The framework will verify that the return status code is 200 (default)

You can run this test by doing: python tapi.py

Thus, without writing a single line of code, you have successfully verfied that this endpoint works.

CHAPTER 6

User Guide

Unit tests in TAPI are specified via a json file. i.e. You specify the input and describe what the output should look like. (Optionally specifying some confirmation steps as well)

The default name of the json file is tapi.json. However, you could name it something diferrent and run it like:

python tapi.py -c <my_new_config>.json

By default the above will also verify that the provided json file has the correct format and all required key/values are present etc. If you only wish to verify the json file and not run any tests:

python tapi.py -c <my_new_config>.json -v

You can also specify which test you wish to run by providing their ids: (default is to run all tests)

python tapi.py -c <my_new_config>.json -i id1,id2,id3

See python tapi.py –help for more details.

6.1 Fundamentals

The fundamental elements in a tapi.json file is the request/response elements. A request element looks like:

```
{
    "request": {
        "url": "<some url>",
        "verb": "get",
        "headers": {
            "header_key1": "header_value1",
            "header_key2": "header_value2",
        },
        "payload": {
            "payload_key1": "payload_value1",
            "payload_key2": "payload_value2",
        }
    }
}
```

The only required field in the above is the ‘url’ field. The above snippet will cause a GET request to be made to the url <some url> along with headers as in the field ‘headers’ and the ‘payload’ form encoded.

The ‘verb’ defaults to GET. Other possible values are POST/PUT/DELETE

A response element looks like:

```
{
    "response": {
        "status_code": 200,
        "headers": {
            "header_key1": "header_value1",
            "header_key2": "header_value2"
        },
        "body": {
            "$.name": "foobar"
        }
    }
}
```

The response subsection is typically optional. Add it if you need to check something other than the defaults. The default status_code is 200. The block causes the framework to verify that the response received after making the HTTP request (as defined by the 'request' section) matches the values specified. If it does not, a failure is reported.

6.2 Primary Elements

There are 4 main primary subsections in a test:

1. startup
2. main
3. confirm
4. teardown

A startup section is a list of main/confirm subsections. These are run in order before any test. The idea is to do any setup like functionality in these blocks. It is always optional.

A main subsection details the api being tested. It is the only required subsection in any test.

A confirm subsection typically is an api call to confirm that the api being tested actually did the work. e.g. if you are testing any api to add a user, a confirm subsection could do a GET on that user to ensure that the user was actually added. It is always optional.

A teardown is run at the end of each test. It is to perform any possible cleanup action you may have. it is always optional.

6.3 Overall Structure

The tapi.json file has the following overall structure:

```
{
  "heading": "Some heading for what you are testing",
  "base_url": "http://api.example.com",
  "common": {...},
  "on_failure": "continue",
  "startup_harness": [...],
  "teardown_harness": [...],
  "tests": [...]
}
```

The field descriptions are as follows:

Field	Type	Default	Required	Description
heading	string	None	No	a human friendly name for this test suite
base_url	string	None	No	the base url to prepend to every url path specified in the tests
common	dict	{}	No	common parameters for requests/response
on_failure	string	continue	No	action in case of failure. (continue/abort)
startup_harness	list	[]	No	actions to perform before the run begins
teardown_harness	list	[]	No	actions to perform after the test run
tests	list	[]	Yes	tests to be run

The following subsections will detail the individual compound keys. (ROOT indicates the root of the json structure.)

6.3.1 ROOT:common

This element contains all the request params that will be common to all requests made by Tapi. It can also contain a response section that can contain all data that needs to be validated. Here is a sample:

```
{
    "main": {
        "request": {
            "headers": {
                "accept-encoding": "compress, gzip"
            }
        }
        "response": {
            "status_code": 200,
            "headers": {
                "content-type": "application/json"
            }
        }
    }
}
```

The above snippet will send the accept-encoding header to 'compress, gzip' in every request made by Tapi (unless overidden in the test config itself). When a response arrives, it will check if the status code is 200 and the response header content-type is set to 'application/json'. Else a failure is recorded.

6.3.2 ROOT:on_failure

This determines what should be done in case a test fails. Possible values are 'continue' and 'abort'

6.3.3 ROOT:startup_harness

These are API calls that are made before the test run. It's called only once during the entire run at the beginning. A sample harness is as follows:

```
[{
    "main": {
        "request": {
            "url": "http://api.example.com/init1"
        }
    }
},
{
    "main": {
        "request": {
```

```
            "url": "http://api.example.com/init2"
        }
    }
},
{
    "main": {
        "request": {
            "url": "http://api.example.com/init3"
        }
    }
}
]
```

Note that the response status code is verified by default. If any request fails, the tests do not begin.

6.3.4 ROOT:teardown_harness

These are API calls that are made after all tests run. It's essentialy a list of 'ROOT:common' sections. It's called only once during the entire run at the end. A sample harness is as follows:

```
[{
    "main": {
        "request": {
            "url": "http://api.example.com/cleanup1"
        }
    }
},
{
    "main": {
        "request": {
            "url": "http://api.example.com/cleanup2"
        }
    }
},
{
    "main": {
        "request": {
            "url": "http://api.example.com/cleanup3"
        }
    }
}
]
```

Note that the response status code is verified by default. If any request fails, the test run is indicated as a failure.

6.3.5 ROOT:tests

This is essentially the meat of the framework. It's where all the requests to test each endpoint is specified. It contains a list of sections wherein each section specifies how an endpoint should be requested and how the response should be verfied. Here is an example of a individual test:

```
{
    "name": "new user",
    "id": "new_user",
    "startup": [
        {
```

```
            "main": {
                "request": {
                    "url": "/startup",
                    "verb": "post"
                }
            }
        }
    ],
    "main": {
        "request": {
          "url": "/endpoint",
          "verb": "post",
          "payload": {
            "name": "bob",
            "age": "20"
          }
        },
        "response": {
            "status_code": 201,
            "headers": {
                "auth-token": "*"
            }
            "body": {
                "$.name": "bob"
            }
        }
    },
    "confirm": {
        "main": {
            "request": {
                "url": "/endpoint/[[self._.response.body.name]]"
              },
              "response": {
                "body": {
                  "$.name": "bob",
                  "$.age": "20"
                }
            }
        }
    },
    "teardown": [
        {
            "main": {
                "request": {
                    "url": "/teardown",
                    "verb": "post"
                }
            }
        }
    ]
}
```

The field descriptions are as follows:

Field	Type	Default	Required	Description
name	string	None	Yes	a human friendly name for the test
id	string	None	No	unique id to identify this test.
startup	list	None	No	list of endpoints to call before running the test
teardown	list	None	No	list of endpoints to call after running the test
request	dict	None	Yes	request object. Possible keys are: url - url to tes
response	dict	None	No	response object to be verified. Possible keys are
confirm	dict	None	No	confirm that the API request just worked. It cor

6.4 Tapi Expressions

In so happens sometimes that the values of fields need to be computed and are not readily available. e.g. we may want to add an 'authorization' header with has the sha256 hash of the username:password or something similar. In order to facilitate such situations, you can use Script Tapi Expressions. These are essentially python scripts that take some parameters and output a result.

There are three types of Script Tapi Expressions, one for requests, one for responses and one for tokens. The one for requests always returns a value which is used in the HTTP request. The script for response is one that takes some args and always returns True/False i.e. telling us if the response matched the spec or not.

A sample request tapi expression is like:

body -{dict of key/value pairs that need to match (Optional) Use the

```
    "request": {
        "main": {
            "url": "http://www.api.example.com",
            "id": "some_id",
            "verb": "post",
            "headers": {
                "authorization": "[[script:request_some_id_authorization.py]]"
            },
            "payload": "[[script:request_some_id_payload.py]]"
        }
    }
}
```

The framework will then look for the script request_some_id_authorization.py/request_some_id_payload.py in the same directory as the tapi.json (or whatever you have named your config as) and execute it. The return value of the script is then used when making the HTTP request.

The request scripts should have this general form:

```
class RequestRunner(object):

    @classmethod
    def get_value(cls, test_output_so_far, config_data):

        #do whatever you need here...

        return 'some value'
```

The framework will import RequestRunner and calls method get_value. The argument test_output_so_far is a dict of the test request/response results of any previous subsections so far e.g. startup, main subsections etc. The config_data is the json config of the test under consideration. Thus this function has all the info it needs and it can do whatever logic it wants to do and finally return a result to be used as a value in the HTTP request.

Similarly the response tapi script has the form:

```
class ResponseRunner(object):

    @classmethod
    def validate(cls, test_output_so_far, test_config_data, response):

        return len(json.loads(response)) == 2
```

The framework will import ResponseRunner and call method validate. Args provided include the HTTP response object. The script can then do any calculation it neds and finally return a True/False answer.

Sometimes, we would also like to access the value of a previously calculated header in future subsections. e.g. the first time you login, if you get a auth-token. And later in every subsequent API call, you need to specify that auth-token, you can use a token tapi expression:

```
{
    "main": {
        "request": {
            "url": "http://www.api.example.com/login",
            "id": "some_id",
            "verb": "post",
            "payload": {
                "username": "foo",
                "password": "bar",
            }
        },
        "response": {
            "headers": {
                "auth-token": "*"
            }
        }
    },
    "confirm": {
        "request": {
            "url": "http://www.api.example.com/dashboard",
            "headers": {
                "auth-token": "[[token:main.response.headers.auth-token]]"
            }
        }
    }
}
```

The [[token:main.response.headers.auth-token]] tells the framework that the auth-token value should be the same as in the response header from the initial request. Note the asterix in the first api response. This tells the tapi framework to only check for the existence of the key, any value returned by the server is ok.

Finally, you can also access environment vaiables by using a tapi expression like [[env:$USERNAME]] - this will be replaced by the value of the $USERNAME environment variable.

CHAPTER 7

Examples

7.1 Simple1

```
{
    "heading": "simple1 example",
    "tests": [{
        "main": {
            "request": {
                "url": "http://api.example.com/users"
            }
        }
    }]
}
```

The above makes a GET(default verb) request to the url and the response is verified to have a status code of 200 (default)

7.2 Simple2

```
{
    "heading": "simple2 example",
    "tests": [{
        "main": {
            "request": {
                "url": "http://api.example.com/users",
                "verb": "get"
            },
            "response": {
                "status_code": 200
            }
        }
    }]
}
```

The same as Simple1, it just makes the verb and status code checks explicit.

7.3 Simple3

```
{
    "heading": "simple3 example",
    "tests": [{
        "main": {
            "request": {
                "url": "http://api.example.com/users",
                "verb": "post",
                "headers": {
                    "accept-encoding": "compress, gzip"
                },
                "payload": {
                    "name": "bob"
                }
            },
            "response": {
                "status_code": 201
            }
        }
    }]
}
```

This makes a POST request to the endpoint (api.example.com/users) with POST payload name=bob. It also adds a request header of Accept-Encoding as 'compress, gzip'. Response status_code is verified to be 201.

7.4 Simple4

```
{
    "heading": "simple4 example",
    "tests": [{
        "main": {
            "request": {
                "url": "http://api.example.com/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip"
                }
            },
            "response": {
                "status_code": 200,
                "headers": {
                    "content-type": "application/json"
                },
                "body": {
                    "$.name": "bob"
                }
            }
        }
    }]
}
```

GETs user bob details and ensures that the response header of Content-Type is set to application/json and response body is a dict which has key/value pair name/bob.

7.5 Intermediate1

```
{
    "heading": "intermediate1 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
              }
            }
        }
    },
    "on_failure": "abort",
    "tests": [{
        "main": {
            "request": {
                "url": "/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip"
                }
            },
            "response": {
                "body": {
                    "$.name": "bob"
                }
            }
        }
    }]
}
```

Similar to Simple4. It adds a base_url to the global 'common' section which will prepend base_url to every url parameter. Further it also says that every response should have a 200 status code and the Content-Type header should be set to application/json. Thus they have been removed from the tests[0]/response section as it is now redundant. It also specifies the action of 'abort' in case any test fails.

7.6 Intermediate2

```
{
    "heading": "intermediate2 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
              }
            }
        }
    },
    "on_failure": "abort",
```

```
    "startup_harness": [{
        "main": {
            "request": {
                "url": "/init"
            }
        }
    }],
    "teardown_harness": [{
        "main": {
            "request": {
                "url": "/cleanup"
            }
        }
    }],
    "tests": [{
        "main": {
            "request": {
                "url": "/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip"
                }
            },
            "response": {
                "body": {
                    "$.name": "bob"
                }
            }
        }
    }]
}
```

Same as Intermediate1, but it also specifies a startup/teardown action at the very begining and very end of the test run.

7.7 Advanced1

```
{
    "heading": "advanced1 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
              }
            }
        }
    },
    "on_failure": "abort",
    "startup_harness": [{
        "main": {
            "request": {
                "url": "/init"
            }
        }
```

```
    }],
    "teardown_harness": [{
        "main": {
            "request": {
                "url": "/cleanup"
            }
        }
    }],
    "tests": [
        {
            "main": {
                "request": {
                    "url": "/users",
                    "verb": "post",
                    "payload": {
                        "name": "bob",
                        "age": 20
                    }
                },
                "response": {
                    "status_code": 201
                }
            },
            "confirm": {
                "request": {
                    "url": "/users/bob",
                    "verb": "get",
                    "headers": {
                        "accept-encoding": "compress, gzip"
                    }
                },
                "response": {
                    "body": {
                        "$.name": "bob"
                    }
                }
            }
        }]
}
```

This example runs a test that has two critical parts: main and confirm. It posts data to the /users endpoint, verifies that the response is OK and then confirms that the API did indeed do what it said it was going to do by GETing the newly created resource.

7.8 Advanced2

```
{
    "heading": "advanced2 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
```

```
            }
        }
    }
},
"on_failure": "abort",
"startup_harness": [{
    "main": {
        "request": {
            "url": "/init"
        }
    }
}],
"teardown_harness": [{
    "main": {
        "request": {
            "url": "/cleanup"
        }
    }
}],
"tests": [
    {
        "main": {
            "request": {
                "url": "/users",
                "verb": "post",
                "payload": {
                    "name": "bob",
                    "age": 20
                }
            },
            "response": {
                "status_code": 201
            }
        },
        "confirm": {
            "request": {
                "url": "/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip"
                }
            },
            "response": {
                "body": {
                    "$.name": "bob"
                }
            }
        }
    },

    {
        "main": {
            "request": {
                "url": "/users",
                "verb": "post",
                "payload": {
                    "name": "jane",
                    "age": 30
```

```
                    }
                },
                "response": {
                    "status_code": 201
                }
            },
            "confirm": {
                "request": {
                    "url": "/users/jane",
                    "verb": "get",
                    "headers": {
                        "accept-encoding": "compress, gzip"
                    }
                },
                "response": {
                    "body": {
                        "$.name": "jane"
                    }
                }
            }
        }
        ]
}
```

Similar to Advanced1 but shows that you can add as many tests as you like because 'tests' is a list. In the above example we add a new user jane and verify that she has been added too.

7.9 Advanced3

```
{
    "heading": "advanced3 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
              }
            }
        },
        "startup": [
            {
                "main": {
                    "request": {
                        "url": "/start_timer"
                    }
                }
            }
        ],
        "teardown": [
            {
                "main": {
                    "request": {
                        "url": "/end_timer"
                    }
```

```
            }
        }
    ]
},
"on_failure": "abort",
"startup_harness": [{
    "main": {
        "request": {
            "url": "/init"
        }
    }
}],
"teardown_harness": [{
    "main": {
        "request": {
            "url": "/cleanup"
        }
    }
}],
"tests": [
    {
        "main": {
            "request": {
                "url": "/users",
                "verb": "post",
                "payload": {
                    "name": "bob",
                    "age": 20
                }
            },
            "response": {
                "status_code": 201
            }
        },
        "confirm": {
            "request": {
                "url": "/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip"
                }
            },
            "response": {
                "body": {
                    "$.name": "bob"
                }
            }
        }
    },

    {
        "main": {
            "request": {
                "url": "/users",
                "verb": "post",
                "payload": {
                    "name": "jane",
                    "age": 30
```

```
                    }
                },
                "response": {
                    "status_code": 201
                }
            },
            "confirm": {
                "request": {
                    "url": "/users/jane",
                    "verb": "get",
                    "headers": {
                        "accept-encoding": "compress, gzip"
                    }
                },
                "response": {
                    "body": {
                        "$.name": "jane"
                    }
                }
            }
        }
        ]
}
```

Similar to Advanced2, but here we also specify a global startup/teardown section. This will get called before **each** test run. (Note that the startup_harness/teardown_harness are called only once in their lifetime)

7.10 Advanced4

```
{
    "heading": "advanced4 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
              }
            }
        },
        "startup": [
            {
                "main": {
                    "request": {
                        "url": "/start_timer"
                    }
                }
            }
        ],
        "teardown": [
            {
                "main": {
                    "request": {
                        "url": "/end_timer"
                    }
```

```
                }
            }
        ]
    },
    "on_failure": "abort",
    "startup_harness": [{
        "main": {
            "request": {
                "url": "/init"
            }
        }
    }],
    "teardown_harness": [{
        "main": {
            "request": {
                "url": "/cleanup"
            }
        }
    }],
    "tests": [
        {
            "main": {
                "request": {
                    "url": "/users",
                    "verb": "post",
                    "payload": {
                        "name": "bob",
                        "age": 20
                    }
                },
                "response": {
                    "status_code": 201
                }
            },
            "confirm": {
                "request": {
                    "url": "/users/bob",
                    "verb": "get",
                    "headers": {
                        "accept-encoding": "compress, gzip"
                    }
                },
                "response": {
                    "body": {
                        "$.name": "bob"
                    }
                }
            }
        },

        {
            "main": {
                "request": {
                    "url": "/users",
                    "verb": "post",
                    "payload": {
                        "name": "jane",
                        "age": 30
```

```
                    }
                },
                "response": {
                    "status_code": 201
                }
            },
            "confirm": {
                "request": {
                    "url": "/users/jane",
                    "verb": "get",
                    "headers": {
                        "accept-encoding": "compress, gzip"
                    }
                },
                "response": {
                    "body": {
                        "$.name": "jane"
                    }
                }
            },
            "startup": [
                {
                    "main": {
                        "request": {
                            "url": "/start_jane_timer"
                        }
                    }
                }
            ],
            "teardown": [
                {
                    "main": {
                        "request": {
                            "url": "/stop_jane_timer"
                        }
                    }
                }
            ]
        }
        ]
}
```

Similar to Advanced3, but now user jane has her own custom startup/teardown section. This shows that the global parameters can be overridden within the test very easily.

7.11 Advanced5

```
{
    "heading": "advanced5 example",
    "base_url": "http://api.example.com",
    "common": {
        "main": {
            "response": {
              "status_code": 200,
              "headers": {
                "content-type": "application/json"
```

```
            }
        }
    },
    "startup": [
        {
            "main": {
                "request": {
                    "url": "/start_timer"
                }
            }
        }
    ],
    "teardown": [
        {
            "main": {
                "request": {
                    "url": "/end_timer"
                }
            }
        }
    ]
},
"on_failure": "abort",
"startup_harness": [{
    "main": {
        "request": {
            "url": "/init"
        }
    }
}],
"teardown_harness": [{
    "main": {
        "request": {
            "url": "/cleanup"
        }
    }
}],
"tests": [
    {
        "id": "postuser",
        "startup": [
            {
                "main": {
                    "request": {
                        "url": "/login",
                        "verb": "post",
                        "payload": {
                            "name": "[[env:$USERNAME]]",
                            "password": "[[env:$PASSWORD]]"
                        }
                    },
                    "response": {
                        "status_code": 200,
                        "headers": {
                            "auth-token": "*"
                        }
                    }
                }
```

```
            }
        ],
        "main": {
            "request": {
                "url": "/users",
                "verb": "post",
                "payload": {
                    "name": "bob",
                    "age": 20,
                    "bank": "[[script:request_postuser_bank.py]]"
                },
                "headers": {
                    "auth-token": "[[token:startup[0].main.response.headers.auth-token]]"
                }
            },
            "response": {
                "status_code": 201,
                "body": "[[script:response_postuser_body.py]]"
            }
        },
        "confirm": {
            "request": {
                "url": "/users/bob",
                "verb": "get",
                "headers": {
                    "accept-encoding": "compress, gzip",
                    "auth-token": "[[token:startup[0].main.response.headers.auth-token]]"
                }
            },
            "response": {
                "body": {
                    "$.name": "bob"
                }
            }
        }
    }
    ]
}
```

Similar to Advanced4, but here we begin everything with a test to /login. Note that we also assigned that test an id. We ensured that the response header has 'auth-token' (but don't worry about it's content, hence the star). In later tests, we want to send in the same auth-token in every request and we do this by aceesing the original auth-token via the tapi expression i.e. startup[0].main.response.headers.auth-token. (remember to enclose it in [[]]). Also notice the [[env:$USERNAME]]. This means that the value of the environment variable $USERNAME is used here. The final point to notice is the ability to run arbitrary python scripts to either get some input value or verify some response result e.g. "bank": "[[script:request_postuser_bank.py]]". This means that the bank input parameter will be populated with the output of the script request_postuser_bank.py. Similarly "body": "[[script:response_postuser_body.py]]" means that the script response_postuser_body.py will be called and it's output should be True/False to indicate if it passed the check. Both scripts will receive the current unit test parametes as a json input. The convention is to name the python script as [requestlresponse]_<id>_<field>.py. This way it will be easy to identify which test/field this script pertains to.

CHAPTER 8

Source

https://github.com/jimmyislive/tapi

Follow me @jimmyislive

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*