# Apitax

Pronounced: *ahhp-ehh-tax*

**tl;dr: Code examples are at the very *bottom* of this documentation; however, I highly suggest you read through the documentation to learn what is possible.**

Finally, as Apitax features an exponential amount of various interactions, not all of them are documented here. Experiment with the syntax, Learn from the syntax, and Enjoy Apitax!

## What is it

Apitax is an API automation framework utilizing Commandtax and Scriptax. Commandtax is an API language which helps to quickly prototype powerful rest API requests. Scriptax is an automation language which utilizes Commandtax.

A more winded (and more detailed!) description called **Why use Apitax** is at the very bottom of this documentation.

## Builds and Installation

### Pip

Apitax on PyPi: https://pypi.org/project/apitax/

Using pip, you can bring in Apitax to your project. However, there are a couple of other commands you need to run to ensure Apitax is ready to go.

* First, install Apitax using pip: pip install apitax
* Then, create a small Script that calls Apitax with some predefined parameters. ```python3 [setup\_apitax.py]

from apitax import Apitax

def main(): apitax = Apitax() apitax.apitax(["--build-only", "--debug"])

if **name** == '**main**': main() ``\* Execute that script:python setup\_apitax.py` \* This can take several minutes to complete. Please be patient.

### Docker

You can use the provided Docker file and requirements file to quickly bake your project into a container with Apitax. To do this, follow these steps. \* Create a new folder, I'm going to call it myimage, but you can call it whatever you like. \* Inside of the myimage folder, create a sub-folder called app. \* Place all of your application and project files inside of the app sub-folder \* Back inside of the myimage folder, copy in the Dockerfile and requirements.txt file found inside of the docker directory in this repo. \* Feel free to modify the Dockerfile and requirements.txt to suit your projects needs. \* You should now have a folder called myimage which contains Dockerfile, requirements.txt, and a sub-folder called app containing all of your application/project code. \* Ensure you have navigated to the myimage directory. This should be the directory where you execute the next few commands from. \* Build the image: docker build --no-cache -t my-amazing-image . \* Run the newly created image: docker run -d -p 5080:5080 my-amazing-image \* This will run the container in the background, to run the container in the foreground remove the -d flag.

### Jenkins

* Jenkins is available here: https://openrubicon.com/blue/organizations/jenkins/Apitax/activity
* This address may change in the future
* Builds are not available on the Jenkins platform, and they likely never will be
* Builds are triggered periodically or via a Git Push
* Build artifacts will likely be pushed to a downloads server in the near future
  + Only non-dependency versions will be offered. On average, these are about 7.5% of the size of the versions which includes dependencies
  + Major and minor releases will also feature a dependency build, however bug and security releases will not feature a dependency version
  + It is best practice to always download the no-deps version and ensure you download the dependencies yourself

### Packaging Instructions

* Use either Powershell or Bash
* Building the package: python setup.py sdist bdist\_wheel
* Upload the package to Pypi: twine upload dist/\* -r pypi
* More information can be found here: https://gist.github.com/ShawnClake/759e9d09af868ef18f8c7b39d1684ad4

### Readme Conversion Instructions

* Install pandoc if it is not already installed: sudo apt-get install pandoc
* Run the command: pandoc -o readme.docx -f markdown -t docx README.md

### Compile Antlr Grammar from another directory

This is sometimes necessary due to a bug in the antlr compiler with regards to paths \* Download the antlr compiler .jar file and save it somewhere. \* Inside of the directory where the .jar was saved, create the following folder hierarchy \* build \* src \* scripts \* logs \* Create a .bat or .sh file (dependent on OS) and add the following script:

#!/bin/bash  
grammardir=~/grammar  
apitaxdir=~/apitax-dev/Apitax/apitax/grammar  
antlr='antlr-4.7.1-complete.jar'  
java -jar $grammardir/$antlr -Dlanguage=Python3 $grammardir/src/$2  
java -jar $grammardir/$antlr -lib $grammardir/src -o $grammardir/build -listener -visitor -Dlanguage=Python3 $grammardir/src/$1  
cp -r $grammardir/build/\* $apitaxdir/build  
cp $grammardir/src/$2 $apitaxdir/src  
cp $grammardir/src/$1 $apitaxdir/src

* Anytime you want to make changes to the grammar, do it from the new ~/grammar/src directory and run the script
  + You must pass in your parser and lexer grammars as arguments bash ~/grammar/run.sh Ah210.g4 AhLex210.g4

## Documentation and Usage

### Commandtax - Data Gathering, Manipulating, Usage

#### Existing

* script <pathToSomeScript>
  + Sequences of commands to automate the execution of API requests
  + Scripts can include additional script commands which effectively means scripts can be nested
  + Scripts are run in order from top to bottom, if a nested script is found it executes the nested script before continuing through the current script
* custom <someCustomCommand>
  + Processes a custom request which is not baked into the utility
  + Parameters
    - --get : Uses a get method
    - --post : Uses a post method
    - --put : Uses a put method
    - --patch : Uses a patch method
    - --delete : Uses a delete method
    - --url : (string) The endpoint
    - --debug : Sets debug mode
    - --sensitive : Sets sensitive mode
    - --data-post : (json string) Any post data
    - --data-query : (json string) Any query parameters ie. Endpoint.com/something?this=queryparam
    - --data-path : (json string) Any url path variables ie. Endpoint.com/users/{path\_var}/show

#### Coming Soon

* shell <someCommand>
  + Runs the command in the shell and returns the response from the shell

### Scriptax - Control Flow, Scoping, and Automation

#### Existing keywords

* ct("<someCommand>") {% %}
  + Commandtax execution
  + A command is executed during the parsing of the line and its response is returned
  + Supports async, callbacks
* get("<someCommand>", {dataObj}, param1...n) {% %}
  + Executes a get request
  + Data Object takes optional keys which correspond to Commandtax custom data: post, query, path, header
  + Returns the result
  + Supports async, callbacks
* put("<someCommand>", {dataObj}, param1...n) {% %}
  + Executes a put request
  + Data Object takes optional keys which correspond to Commandtax custom data: post, query, path, header
  + Returns the result
  + Supports async, callbacks
* patch("<someCommand>", {dataObj}, param1...n) {% %}
  + Executes a patch request
  + Data Object takes optional keys which correspond to Commandtax custom data: post, query, path, header
  + Returns the result
  + Supports async, callbacks
* post("<someCommand>", {dataObj}, param1...n) {% %}
  + Executes a post request
  + Data Object takes optional keys which correspond to Commandtax custom data: post, query, path, header
  + Returns the result
  + Supports async, callbacks
* delete("<someCommand>", {dataObj}, param1...n) {% %}
  + Executes a delete request
  + Data Object takes optional keys which correspond to Commandtax custom data: post, query, path, header
  + Returns the result
  + Supports async, callbacks
* script("<someScriptFile>", {emptyObject}, param1...n) {% %}
  + Executes a script
  + Returns the result
  + Supports async, callbacks
* async
  + Add this keyword in front of any other keyword which supports async to run the operation in a new thread
  + async get("http://placeholderjson.com/users", {}) {% %}
  + someVar = async get("http://placeholderjson.com/users", {}) {% %}
  + Callbacks execute prior to storing into variable
  + Variable will be initialized with a thread instance and once the thread completes it will be replaced with the result as returned via the callback
* await <someOptionalVar>;
  + When a variable is specified, wait until the async execution specified by that variable completes.
  + When no variable is specified, wait until all of the open threads in the current script complete before moving on
* {% %};
  + Callback block
  + The contents will be executed in an isolated scope, usually only having access to a results variable
* str()
  + Cast to string
* int()
  + Cast to rounded integer
* dec()
  + Cast to float
* bool()
  + Cast to true/false
* list()
  + Cast to list
  + Only works on strings
* dict()
  + Cast to dictionary
  + Works on lists, strings, ints, decs
* #
  + Return the length of the variable
  + Works on strings, lists, dictionaries (Only returns the top level count)
* del <someVar>
  + Remove someVar from the current scope
* return <optionalVar>
  + Exits the script immediately and returns some expression
* options
  + Used to specify options for the script
* sig param1Required, thisParam=isOptional, thisOneisRequired;
  + Specify parameters for a script
* if (condition) {}
  + IF statement
* while (condition) {}
  + While loop
* for <someVar> in <existingVar> {}
  + Loop through each item in a list in order
* for <someVar> in <someNumber> {}
  + Loop from 1 to someNumber and set someVar to the current iteration
* each <someList> {% %};
  + Loop through a list setting results to the current item and executing instructions in an isolated callback
* <someVar> = <someValue>
  + Sets a variable
  + Supports expressions, strings, numbers, booleans, dictionaries, lists, and commandtax responses
* "this is a string {{ someVar }}"
  + Injects the contents of a variable
  + Fancy stuff is possible such as: set newVar = ct("{{someVar}}")
* {{ r: someResponse }}
  + Injects the response of some request
* import ct("some commandtax")
  + Executes a command and imports the response to the current scope
* export ct("some commandtax"), export someVar
  + Imports the values to the current scope and exports them to allow a parent scope to access these values
* name <someName>
  + Sets the reference name of the script.
  + Supports strings and expressions
* url <someUrl>
  + Sets the current working URL to be used in further commandtax
* log("log some output to the console & log file")
  + Supports expressions
* // some comment
  + Inline comment
* /\* some comment spanning multiple lines \*/
  + Block comment

#### Best Practices

* Start each script with an options line immediately followed by a name line.
* End each script with await; to ensure all async requests are completed before it returns to the parent script.
  + Ending with await; followed by return \<someVar\>; is also acceptable
* Delete unused variables from the root scope when they are no longer needed del \<someVar\>
* Only export/return necessary variables from subscripts
* Keep the root scope as clean as possible
* Scripts should be small, containerized pieces of code that strictly follow SRP.
  + Think of Scripts as lego blocks, eventually you can put a bunch of them together to build something really cool
* Scriptax is a fairly flexible, forgiving, and powerful language
  + Play around to see what you can and cannot do. There are too many edge cases to list explicitly.

#### Coming Soon

* A time method

#### Tidbits

* You can use arrays via dot notation
  + set someVar.1 = num1
  + set someVar.2 = num2
  + set someVar.{{counter}} = num3
  + When doing this, the first usage of a variable must either be someVar = "{}" or an index as part of that object. Failure to do this will result in errors being thrown.

### Commandline Interface (CLI)

* You can activate Apitax from the CLI directly without needing a wrapper package
* Run the tests.py file found in the root Apitax directory, and supply the following arguments
  + --cli : (Optional) Quickly select CLI mode
  + -- web : (Optional) Quickly select web server mode
  + --debug : (Optional) Output the request response status, headers, and body
  + --no-build : (Optional) Will not rebuild the UI assets on launch
  + -u : (Optional) Specify the authentication username - Only applicable in CLI mode
  + -p : (Optional) Ask for password input right away. If -u is specified, but this is not, the application will ask for a new set of credentials for authentication. But, it will use the -u value for any username fields within further requests. This allows someone to authenticate as admin, but run commands applicable to another user.
  + -r : (Optional) The request - Only applicable in CLI mode
  + -s : (Optional) Run a script of requests specified by the file path , authentication for all of these requests are specified by -u & -p

### Modes

* CLI : Make a request from the command line
* WEB : Start the web server interface
* Grammar Test : Run a test of the parsing and grammars

### Configuration

* An example configuration file is stored in the repo root

### Supported Authentication

* Authentication is prebuilt for HTTP Basic and Token based authentication
* Authentication is largely left up to a developer in custom scenarios
  + Driver files facilitate this requirement

### Drivers and Plugins

\*\* This has been redone and requires updated configuration \*\* \* Drivers and plugins are used to extend the functionality of Apitax to an arbitrary API \* While dynamically injecting drivers and plugins is on the todo list, for now all plugins must go into the apitax/drivers/plugins directory \* Each driver requires at least a core plugin file \* This file describes any custom parameters with regards to authentication, endpoints, and the requirements of the specific API the driver is built for \* Optionally, a driver can also include commandtax plugins which are used to specify shortcut commands \* To do this, a core commandtax plugin file is required as well as a commandtax plugins directory. The core commandtax plugin file must contain the suffic Commands \* Core commandtax plugins file: apitax/drivers/plugins/commandtax/CoreFileCommands.py \* Core commandtax plugins directory: apitax/drivers/plugins/commandtax/<theNameofTheDriver>/ \* Inside of this new plugins directory, you can create shortcut files to route specialized commands. You can see examples of this in the apitax source code with regards to the ApitaxTests driver and plugin files \* Shortcut commands are really just scripts which are aliased to a command

### Examples of Apitax in Action:

#### Commandtax Examples

custom --get --url <someEndpoint>  
custom --get --url <someEndpoint> --data-param '{"is\_domain": true}'  
custom --post --url <someEndpoint> --data-post '{"title": "im the title"}'  
custom --put --url <someEndpoint>   
custom --patch --url <someEndpoint>  
custom --delete --url <someEndpoint>   
custom --get --url <someEndpoint> --data-param '{"user.id": "1"}'  
custom --get --url <someEndpoint>/with/some/{ohyear}/url/params/{981} --data-param '{"is\_domain": true}' --data-path '{"ohyeah":"no", "981": "yes"}'

script ~/path/to/my/script.ah

#### Scriptax Examples

// async-tests.ah  
  
url "https://jsonplaceholder.typicode.com";  
  
bob = [];  
threads = [];  
  
/\*each get("/users", {})  
{%  
 url "https://jsonplaceholder.typicode.com";  
 threads[] = async get("/posts", {  
 "query": {  
 "userId": result.id,  
 },  
 }) {%  
 log("The user has these posts: " + result);   
 %};  
%};\*/  
  
for user in get("/users", {})  
 threads[] = async get("/posts", {  
 "query": {  
 "userId": user.id,  
 },  
 }) {%  
 log("The user has these posts: " + result);  
 //bob[] = result;  
 %};  
  
await threads;  
  
del threads;  
  
script("apitax/grammar/scripts/base.ah");  
  
log("yo man im at the end");  
  
  
/\*for user in get("/users", {})  
 threads = async get("/posts", {  
 "query": {  
 "userId": user.id,  
 },  
 }) {%  
 log("The user has these posts: " + result);  
 bob[] = result  
 %};  
  
//bob[] = "hey";  
  
for t in threads  
 await t;  
  
log(bob);  
  
log("Im quite confused");\*/

// base.ah  
  
url "https://jsonplaceholder.typicode.com";  
for user in get("/users")  
{  
 /\*result = get("/posts", {  
 "query": {  
 "userId": user.id,  
 }  
 });  
 log("The user has these posts: " + result);\*/  
  
 log("the user has ID: " + user.id);  
  
 async get("/posts", {  
 "query": {  
 "userId": user.id,  
 }  
 }) {%  
 log("The user has these posts: " + result);  
 %};  
  
}   
   
response = post("/posts", {  
 "post": {  
 "title":'foo',  
 "body":'bar',  
 "userId":1  
 }  
});  
  
log(script("apitax/grammar/scripts/jen.ah", {}, "i am parameter 1", "i am parameter 2"));  
  
//response = ;  
  
log("Please " + ct("tests my script", {}, "i should be a parameter"));  
  
  
testlist = ["one", "two", 'three', ["a", 'b', 'c', 'd'], {"test": "failed", "yes": "no"}];  
  
log(#testlist.0);  
  
  
/\*async ct("some command execution")  
{  
log("some integrated callback");  
}  
  
async ct("some command execution");  
  
status = async ct("some command execution");  
  
await status;  
  
status = [firstAsync, secondAsync];  
await status;  
  
async bobo in get("/users")  
{  
 log("Async callback: " + bobo);  
}\*/  
  
async get("/users");  
  
  
  
somelabel = async get("/users")  
{%  
 log("I am an optional callback");  
%};  
//anotherRequest0 = async get("/users");  
//anotherRequest1 = async get("/users");  
//anotherRequest2 = async get("/users");  
  
await somelabel;  
  
log("i am that label" + somelabel);  
  
  
get("/users") {%  
  
log(result);  
  
%};  
  
async get("/users") {%  
  
log(result);  
  
%};  
  
  
for iter in 10 {  
  
 async get("/posts", {  
 "query": {  
 "userId": iter,  
 }  
 }) {%  
 log("The user has these posts: " + result);  
 %};  
  
}  
  
log("here is where the magic happens");  
  
get("/posts", {  
 "query": {  
 "userId": 5,  
 }  
 }) {%  
 log("The user has these posts: " + result);  
%};  
  
  
await;

// jen.ah  
  
options dict('{"params": ["name", "game"]}');  
answer=2+2;  
name "quinn";  
  
set qwer = "im in jens script";  
  
  
export qwer;  
  
//export ct("script apitax/grammar/scripts/shawn.ah");  
  
set iam.hope.this.works = ct("custom --get --url https://jsonplaceholder.typicode.com/users");  
  
export iam;  
  
set test = "hello";  
  
//return '{"cool": "beans"}';  
  
log(params.passed.0);  
  
log(params.passed.1);  
  
log(params.name);  
  
log(params.game);  
  
log(2+2);  
  
return params.passed.1;  
  
//return "test";  
  
  
await;

// my\_script.ah  
  
  
options dict('{"params": ["first"]}');  
  
log("I got here");  
  
log(params.first);  
  
return "work";  
  
  
await;

## Why use Apitax

Let's presume that a backend exists only as an entity that does nothing until it is told to do something.

If this is the case let's also assume there will be no interface into the backend other than a RESTful API.

Okay, so let's say we want to build a frontend that will list every user's 10 most recent posts.

One way we can do this is to make an API request to retrieve a list of our users. Then, in JavaScript, we can loop through that list of users, and for each one we can make an API request to get their most recent posts. In Javascript, we can then compile this list of users together and display it.

Sounds simple - except how do we then speed up this process by utilizing asynchronous code. We begin to introduce all sorts of nested callbacks, desync's in our JavaScript that are complex to sort out, difficult bug testing, and we start to break DRY as we copy bits and pieces of this code to different files/pages to list different types of data in different ways.

We can solve some of our DRY problems by creating giant API classes that have helper methods for each of these types of calls, but it will still become quite a mess.

What we want is to focus on frontend development - not the API request garbage. By doing this, we are enforcing SRP of the frontend architecture as a whole. The bulk of the frontend code should be dedicated to displaying data, getting input, and showing users responses from their input. It should not be thousands of lines of us making API calls and trying to organize the data. There has to be a simpler way, right?

What if the frontend only had 4 different API requests to make. 1 for authentication, 1 for getting a catalog of endpoints, 1 for getting the status of the system, and finally 1 to facilitate the transfer of bidirectional data. Here is where Apitax/Commandtax comes into play. We now have a single endpoint to send a request to, and that request's payload is commandtax.

Great! Now, instead of a bunch of helper methods all over the place, now we might just have some helper string constants that store frequently used commands. Already much cleaner and more workable. But, this still doesn't solve our DRY problem, SRP, or the callback and desync hell we will face from having to make several requests in a row to accomplish a task.

Enter: Scriptax. Now, any of us can create little self-contained sub module files which do these sequential requests for you. These files are called Scripts. One script might have the responsibility of getting a list of users that follows some set of parameters, and another might call that users script and then using the response returned, it could get a list of posts for each of those users and return them.

Okay, now our frontend only has 1 short line of commandtax which is executing a script. If the scripts are written well, there is no more callback hell, SRP is preserved, DRY is enforced, and the data returned to us will already be in a very workable format. Kaboomskies, you just saved yourself 2 dozen lines of hard to debug JavaScript, and instead, it's a one liner that returns the data already in a workable format