pytonik Documentation

Release 1.9.11

pytonik

Mar 29, 2020

Contents

This document refers to version 1.9.11

CHAPTER 1

Pytonik

Pytonik is a python framework built to enhance web development fast and easy, also help web developers to build more apps with less codes. it uses expressive architectural pattern, structured on model view controller MVC and bundles of component to reuse while deploying the framework.

Contents:

1.1 Getting Started

Pytonik framework supports python version from 2.7 to newer, Pytonik can be download from `https://pypi.org/pytonik` using terminal or command line promts `pip install pytonik` or `https://github.com/pytonik/pytonik` The installation will automatically be extracted into `site-packages` in Python directory.

Pytonik provides command line tool that helps in creating project folder structure with the use of `pytonik-start` command, it gives internationalization, `English` as default language and other supported languages are listed below ‘

```
'bg_BG': 'Bulgarian',
'cs_CZ': 'Czech',
'da_DK': 'Danish',
'de_DE': 'German',
'el_GR': 'Greek',
'es_ES': 'Spanish',
'et_EE': 'Estonian',
'fi_FI': 'Finnish',
'fr_FR': 'French',
'hr_HR': 'Croatian',
'hu_HU': 'Hungarian',
'it_IT': 'Italian',
'lt_LT': 'Lithuanian',
'lv_LV': 'Latvian',
'nl_NL': 'Dutch',
'no_NO': 'Norwegian',
```

(continues on next page)

(continued from previous page)

```
'pl_PL': 'Polish',
'pt_PT': 'Portuguese',
'ro_RO': 'Romanian',
'ru_RU': 'Russian',
'sv_SE': 'Swedish',
'tr_TR': 'Turkish',
'zh_CN': 'Chinese',
```

Note: if you receive an error while running the command, the error will stop you from using the command, to get the error fixed, you will be required to add `export LC_ALL=en_US.UTF-8` and `export LANG=en_US.UTF-8` to system path. This error are common on `MAC OS`, `Ubuntu` and `Linux`. if you keep experiencing this error, refer to **File Structure**.

1.1.1 Installations

Install python and make sure it's running on your computer operating system environment or web server.

Python can be downloaded at `https://www.python.org/downloads` If you are using a cloud open terminal type or paste `sudo apt-get install python3` download environment and Pytonik requirement.

During installation process add to path but if missed out, then you will have to add it to environment path once the installation is completed.

After installation process, open Terminal/Command promts and type in `python` or `which python`.

Note:- `which python` command might not work on windows environment, to achieve this, it will need the help of `https://git-scm.com` on windows 10 or newer, install ubuntu `https://www.microsoft.com/en-us/p/ubuntu/9nblggh4msv6`

1.2 Configuration

`Pytonik` supports Common Gateway Interface **(CGI)** and Web Server Gateway Interface **(WSGI)**, it requires configuration to be able to run on Apache or any python **CGI** or **WSGI** enable environment. The Configuration step support both local and cloud server.

Note: Pytonik provides **lite-server**, that helps in deploying pytonik application faster with the use of, `pytonik-start` and `pytonik-server` command, this features was added to **version 1.9.7**, and will be improved in newer versions. You might intend using other technology such as `WSGI` `WAMP`, `XAMPP` etc.

1.2.1 Local Server

Look into configuring `WAMP` , `XAMPP` , `LAMP` and `MAMP`, if you are not using any of the listed local servers don't worry, this process supports **Apache**, both older and newer version, Although you might notice little difference between versions.

Note: If you keep encountering problems during setup or local machine is unable to run Common Gateway Interface **CGI** We reference **pytonik** video tutorials.

Apache

Locate **httpd.conf** in your local server open it with a text or code editor.

Find `AddHandler cgi-script .cgi` add .py `AddHandler cgi-script .cgi .py`.

Find `Options None` change to `Options ExecCGI` or find `Options +Indexes` change to `Options ExecCGI` or change to `Options +Indexes FollowSymLinks +ExecCGI` or find `AllowOverride None` change to `AllowOverride All`.

Find `DirectoryIndex index.html` add index.py `DirectoryIndex index.html index.py`.

Find and uncomment `#` if only is comment with `#LoadModule rewrite_module modules/mod_rewrite.so` and `#LoadModule alias_module modules/mod_alias.so`.

When all the changes are made, save `httpd.conf` file, Restart Apache for changes to take effect.

Server is set and ready to run python CGI application which means you can deploy pytonik on it. Restart server and done.

Virtual Host (vhosts)

Setting up a virtual host is very important espicially when using a local server. virtual host grant you access to own custom domain names within a local environment, instead of `localhost/mypytonik` it uses `mypytonik.test` as the domain name. This features is very important especially when deploying application. Softwares Like `MAMP` and `WAMP` provides **GUI** (graphic user interface) where you can add custom domain names, so you might not have to worry about how to set up **vhosts**, some other software's do not have such features. If you are using `XAMPP` then you will need to confirgure vhost in `httpd-vhosts.conf` file, While `LAMP` uses Terminal for most of its configurations but takes the same proccess. Check the configuration below.

Locate **httpd-vhosts.conf** File

```
<VirtualHost *:80>
        DocumentRoot "c:/xampp/htdocs"
        ServerName localhost
        <Directory "c:/xampp/htdocs">
        </Directory>
</VirtualHost>

<VirtualHost *:80>
        DocumentRoot "c:/xampp/htdocs/mypytonik"
        ServerName mypytonik.test
        <Directory "c:/xampp/htdocs/mypytonik">
        </Directory>
</VirtualHost>
```

1.2.2 Cloud Server

Some Cloud server requires CGI configuration while some have it already configured and enabled. All depends on the company who's offering the services. Look into `CGI` configuration on `CentOS`, `Linux`, `Ubuntu` and terminal will be used for configuration process.

CentOS

Install apache `sudo yum install httpd` open the httpd.conf file `sudo nana /etc/httpd/conf/httpd.conf` now set for editing.

Find `AddHandler cgi-script .cgi` add .py `AddHandler cgi-script .cgi .py`.

Find `DirectoryIndex index.html` add index.py `DirectoryIndex index.html index.py`

When all the changes are made, save httpd.conf file, changes takes effect.

Server is set and ready to run python CGI application which means you can deploy pytonik on it. Restart server `sudo systemctl restart httpd` and done.

Ubuntu/Linux

Install apache `sudo apt-get install apache2`

open the httpd.conf file `sudo nano /etc/httpd/conf/httpd.conf` or `gksu gedit /etc/httpd/conf/httpd.conf` now set for editing.

Find `AddHandler cgi-script .cgi` add .py `AddHandler cgi-script .cgi .py`.

Find `DirectoryIndex index.html` add index.py `DirectoryIndex index.html index.py`

When all the changes are made, save httpd.conf file, changes takes effect.

Server is set and ready to run python CGI application which means you can deploy pytonik on it. Restart server `sudo systemctl restart httpd` and done.

1.3 Dependency

Pytonik is built on python and requires the support of other modules which enhance application development. Without it's dependency functions like database connection, Imaging properties won't work. Below are lists of dependencies:

MYSQL If your web application supports MySQL database this module is required and needed to be installed `pip install mysql-connector`.

POSTGRESQL If your web application supports PostgreSQL database this module is required and needed to be installed `pip install psycopg2`.

ORACLE

If your web application supports ORACLE database this module is required and needed to be installed. `pip install cx-Oracle`.

PILLOW

If your web application support Imaging, resize and other imaging functions this module is required and needed to be installed `pip install Pillow`.

1.4 File Structure

`Pytonik` is a web framework that supports model view controller MVC, web development are made easier and faster using the file structure below, it gives full access to `controller`, `model`, `languages`, `public`, `views` , `environment`. Create folder in your web directory as follows:- Type in `pytonik-start` on Terminal/command or you 'Link to download on github <https://github.com/pytonik/Folder-Structure>'

```
|—MyPytonikApp                          (application folder)
    |— controller
        |— IndexController.py
    |— lang
        |— en.py
    |— model
        |— Mymodel.py
    |— public
        |— Index.py
```

(continues on next page)

(continued from previous page)

```
        |— assets
        |— uploads
        |- .htaccess
    |— views
        |— 404.html
        |— homepage.html
    |- .env
    |- .htaccess
```

Htacess:

It's a configuration file that support pytonik application to run on Apache Server, this file is configured to set index and redirects application Uniform Resource Locator(URL) to public folder.

Controller:

It handles direct regulation of activities within the application, all component, functions and interfacing depends on it.

Lang:

It handles internationalization translation of word or sentence within the application and structured it conventionally.

Model:

It handles structuring each piece of function in the application, components and can be used by the controller or called directly in view files

Public:

It handles asset files such as CSS, JS, images, uploads and access are set to audience

Views:

The result display sent from controller where the end user can see.

Note:- app.log file will be automatically created once pytonik is identified by the application project, the file keeps track of errors and other useful information for bugs fixings.

1.5 Environment File

`Pytonik` environment file contains properties with argument. It is the first step that needs to be taken to setup file and make sure it is configured properly. It handles all `pytonik` application settings that will be used to developed web application and the environment file is saved as `.env`

`Pytonik` framework cannot run or function without setting all necessary environment file argument requirement.

.env Properties :

- route
- dbConnect
- languages
- SMTP
- error

1.5.1 route

route set all default method for reused purposes, either to pass on parameters or revoking controller, below is the illustration. We'll be using `ContactController.py` and `BlogController.py` for demonstrations.

```
'route':
        {
        'edit': 'ContactController@edit',

       'blog': 'blogController@read:id:para',

        }
```

How to defined route:

Custom `'edit': 'ContactController@edit'` edit is function in `ContactController.py` Controller.

Check the result out here, now we will be having:- `http://example.com/edit` instead of `http://example.com/contact/edit`

Pass parameter `'blog': 'blogController@read:id:para'` Where `id` and `para` is an argument and `read` is a function in '' BlogController.py'' Controller. Check the result out here, now we will be having:- `http://example.com/blogs/1/tea` instead of `http://example.com/blog/read/id/1/para/tea`

1.5.2 dbConnect

dbConnect set all Database connection parameters. configure pytonik to work with MYSQL database, parameters are required to enable database to function probably

MYSQL

```
'dbConnect':
         {
              'host': 'localhost',

              'database': 'pytonik-database',

              'prefix': 'prefix_',

              'password': 'database-password',

              'username': 'database-username',

              'port': 'database-port',

              'driver': 'MYSQL'
         }
```

Oracle

```
'dbConnect':
         {
              'host': 'localhost',

              'database': 'pytonik-database',

              'prefix': 'prefix_',
```

(continues on next page)

(continued from previous page)

```
            'password': 'database-password',

            'username': 'database-username',

            'port': 'database-port',

            'driver': 'Oracle'
        }
```

pyPgSQL

```
'dbConnect':
        {
            'host': 'localhost',

            'database': 'pytonik-database',

            'prefix': 'prefix_',

            'password': 'database-password',

            'username': 'database-username',

            'port': 'database-port',

            'driver': 'pyPgSQL'

        }
```

SQLite

```
'dbConnect':
        {
       'path': 'folderDB',

       'prefix': 'prefix_',

       'name': 'databasefile.db'

       'driver': 'SQLite'
        }
```

** Note: ** pytonik driver supports only `SQLite`, ‘‘ MYSQL‘‘, `Oracle` and `PostgreSQL` Database

1.5.3 Languages

‘‘ Pytonik ‘‘ supports internationalize, all languages files are store in lang folder, to enable web application to run multiple language translations, environment file needs to be configured with all necessary argument, below used English as `en` , French as `fr` and Russian as `ru`. The language file is saved as `.py`

Example: `en.py`, `fr.py` , `ru.py`

```
'languages':
{
```

(continues on next page)

(continued from previous page)

```
    'en': 'en',
    'fr': 'fr',
    'ru': 'ru',
}
```

Note: Our web application will be using English as the default language which is en. defined as follow `'default_languages':'en'`

1.5.4 SMTP

To enable application to send mails to or fro, pytonik framework requires ‘‘ SMTP ‘‘ setting as follows.

```
'SMTP':
{
    'server':   'test@server.com',
    'port':     '25',
    'username': 'test@example.com',
    'password': 'testpassword',
}
```

Note: Web application that requires sending of electronic mails, above setting is compulsory.

1.5.5 error

pytonik provides support for error page handling to avoid showing of errors, programing bug or codes to the audience, it handles error such as:- method not found / error in method **400**, page not found **404** and controller not found **405**.

```
'error':
{
    '400':   'custom/error400',
    '404':     'custom/error404',
    '405': 'custom/error405',
}
```

1.5.6 Default

We will look at how to define all default properties as such routes , controllers , actions and languages. Below are examples and you can as well set yours differently.

```
'default_controllers' :'index',
'default_actions': 'index',
'default_routes' : 'index',
'default_languages': 'en'
```

Why `default_controllers` are set to `index` means that the application depends on IndexController, let say your index page is being pointed to IndexController.

Why `default_actions` are set to `index` means all default methods are set to index respective of the controller.

Why `default_routes` are set to `index` means application points to IndexController.

Why `default_languages` set to `en` means application is using en.py language file by default for internationalization.

1.6 Template Engine

Pytonik provides flexible templating engine, making use of `blocks`, `Variables` , `Loops`, `conditionals` `statement` , `operators` , `iteration` and `scopes`.

```
from pytonik.App import App
app = App()

items = [
        dict(name="dog", age=5),
        dict(name="cat", age=2),
        dict(name="snake", age=26),

    ]

data = {
    'items': items,
    'my_var':  "Welcome to Pytonik",
    'my_var_2': "Nothing",
    'status': "active",
    'num': 1,

    }


app.views('index', data)
```

1.6.1 Variables

```
<div>{{my_var}}</div>
```

Block

Type of block if, each and call.

1.6.2 Loops

loop with dictionary or json

```
{% each items %}
    <div>{{it.name}}  {{it.age}}</div>
{% endeach %}
```

loop with `list`

```
{% each [1,2,3] %}
    <div>{{it}}</div>
{% endeach %}
```

Loop items has iteration with a scope, to access attributes which is a parent context or outer variable use `..`

```
{% each items %}
    <div>{{..status}}</div><div>{{it.name}}  {{it.age}}</div>
{% endeach %}
```

1.6.3 Conditionals

Supported operators are: >, >=, <, <=, ==, !=. You can also use conditionals with things that evaluate to truth. `if` conditional statement

```
{% if num %}
    {{my_var}}
{% endif %}
```

if conditional statement with **else**

```
{% if num %}
    {{my_var}}
{% else %}
    {{my_var_2}}

{% endif %}
```

if conditional statement with operator

```
{% if num == 0 %}

    {{my_var}}

{% endif %}
```

1.6.4 Callable

call block, get or passed positional or keyword arguments or parameter. url is class and path is method

```
{% call url 'path' %}
```

url is class and url is method while `path=''` is keyword arguments or parameter

```
{% call url 'url' path='' %}
```

1.7 SMTP

Send email to user with pytonik framework, this could be one part you really want to look at. Pytonik provides module that handles sending of emails and file attachment to users. this module has two methods `send` and `attach` . Before your application can to send or receive email. SMTP settings has to be enabled, to do so, we will work through and see how to configure our web application to send out email to users

SMTP Environment Setting- you might wonder where to get the setting below, you can get smtp settings from your email providers or emailing server, example **GMAIL**, **YAHOO**, **OUTLOOK** or Custom email host.

```
'SMTP':
{
    'server':   'mail.server.com',
    'port':     26,
    'username': 'from@mail.com',
    'password': '231222',
}
```

Import Module

```
from pytonik.Core.SMTP import SMTP
```

Callable .. code-block:: python

mail = SMTP()

Example: `Variable` and `Strings`

```
subject = "My pytonik"
content = "I love Pytonik framework"
from = "from@mail.com"
to = "to@mail.com”
```

Example: Sending Email to user

```
sent = mail.send(from, to, subject, content, header='html')
```

Example: Sending Email with attachment to user

```
file = "my_attachment.pdf"
attached = mail.attach(file).send(from, to, subject, content)
```

Example: Rename file attachment before sending email and attachment to user

```
file = "my_attachment.pdf"
rename = "rename file"
attached = mail.attach(file, rename).send(from, to, subject, content)
```

1.8 Request

Pytonik Request library contains form methods and attributes which comprises of `POST` , `GET` and `PARAM`, it outline how to use form to perform task such as Login, Registering, Posting, Updating, Requesting, Sending actions.

Import Module

```
from pytonik.Request import Request
```

Attribute of Request are :-

- method
- get
- post
- file
- param

method Return `POST` and `GET`

```
Request.method
```

get returns the value of `GET` request by obtaining the value using key attribute.

```
Request.get('name')
```

post returns the value of `POST` request by obtaining the value using key attribute. .. code-block:: python

    Request.post('name')

get param returns the value of GET request by accessing the key attributes to obtain the value from either a custom http query string `https://test.com/id/2/name/dog/age/3` or http request query string `https://test.com/?id=2&name=dog&age=3` . which get attribute cannot perform such tasks

```
Request.param('id')

Request.param('name')

Request.param('age')
```

file returns POST request by accessing the key attributes to obtain the value for sent request from file fields. Request.file('picture')

Example: using post attribute which will return POST Demonstration form with html code

```
<form method="post" enctype="multipart/form-data">
<input type="text"  name="id" value="" >
<input type="text" name="name" value="" >
<input type="text"  name="age" value="" >
<input type="file"  name="picture" value="" >
<button type="submit"  name="submit">Submit</button>
</form>
```

Form actions is handle using the help of operators together with conditional statement to make it easy to check if fields are empty or not :- where `is` is the same as == and `is not` is the same as `!=`. Notice that the form method is set to `POST`

```
if Request.method == "POST":

    id = Request.post('id')
    name = Request.post('name')
    age = Request.post('age')

    picture = Request.file('picture')
    if id == "":
        print("ID is empty")
    elif name == "":
        print("NAME is empty")

    elif age == "":
        print("AGE is empty")

    elif file == "":
        print("PICTURE is empty")
    else:
        print("SUBMITTED successfully")
```

1.9 Session

`pytonik` provide sessions to store data and recall them when needed. Session has property to `set`, `get` and destroy data with in pytonik web framework.

Import Module

```
from pytonik.Session import Session
```

Attribute of Request are :-

- set
- get
- destroy

set initiate data storage over http with keyword argument, `set(key, value="", duration, url, path)`

```
Session.set('key', 'value', 'duration')
```

get retrieve data storage http with keyword argument.

```
Session.get('key')
```

destroy all session over http

```
Session.destroy()
```

destroy session associated with a keyword

```
Session.destroy('name')
```

1.10 File

File upload properties consist of **upload**, **delete**, **ext**, **rename**, **Image**. This process requires installation **PIL** module.
`pip install Pillow`

import Module

```
from pytonik.Core.File import File
```

Callable

```
file = File()
```

How to :

- get file extension
- check file extension
- upload file
- get file size
- check file size
- upload and resize IMAGE

Get file extension

```
ext = file.ext(filename)
```

Check file extension

```
list_ext = ['png', 'jpg', 'JPG', 'jpeg']
ext = file.ext(filename)

if ext in list_ext:
    print("This file is valid")
else:
    print("This file is not valid")
```

Upload file

```
file.upload(thefile, directory, rename)
```

1.10.1 IMAGE

get file size

```
size = file.Image(directory, thefile).size()
```

Check file size

```
custom_size = 1024 * 1024 * 2  * 2MB File Size


size = file.Image(directory, thefile).size()

if custom_size >= size:
    print("Pass File size test")
else:
    print("File Size is greater than its custom size")
```

Upload and resize

```
image = file.Image(directory, thefile)

dimension = {64: 64, 128: 128}

for w, h in dimension.items():
    image.resize(w, h)
```

upload, resize and remain IMAGE - python 2 below

```
image = file.Image(directory, thefile)

dimension = {64: 64, 128: 128}

rename = "Enter the new name of the file"

for w, h in dimension.iteritems():
    image.resize(w, h rename)
```

upload, resize and remain IMAGE - python 3 above

```
image = file.Image(directory, thefile)

dimension = {64: 64, 128: 128}
```

(continues on next page)

(continued from previous page)

```
rename = "Enter the new name of the file"

for w, h in dimension.items():
    image.resize(w, h rename)
```

1.11 Functions

Pytonik Provides bundles of in-bulit functions that support `mvc` app development, Each of the functions are useful and will break all limit and doubt as you get to know how to use them. These modules are created to make app building more easier and faster. Expect more functions in new released versions.

Note: Some functions will be re-consider as standalone module and be made as dependencies in the future. Making pytonik framework more flexible and elegant.

1.11.1 Langauge(__)

pytonik provides inbuilt langauge module __, this module is responsible for translation / internationalization of word around pytonik application. Our project target is to support English, Russain and french world, the we will need to create content with multiple langauges . example when our users access `http://mydomainname.com/en` it will display english content, `http://mydomainname.com/ru` it will display russain content, and `http://mydomainname.com/fr` it will display french content, using the help of `lang` method and `lang` argument.

Note: Language files are stored/saved in `lang` folder as `.py` in pytonik folder project and it uses **dictionary**.

Example: Below will be save in `en.py` file

```
{
    'lng.test': 'Welcome',
    'text' : 'programming'
}
```

Example: Below will be save in `ru.py` file

```
{
    'lng.test': 'zhelannyy',
    'text' : 'programmirovaniye'
}
```

Example: Below will be save in `fr.py` file

```
{
    'lng.test': 'Bienvenue',
    'text' : 'programmation'
}
```

Import Module

```
from pytonik.Functions.__ import __
```

Callable

```
lang = __()
```

Template Engine

```
{% call __ "lng.test" %}

{% call __ lang='lng.test'  %}

{% call __ 'lang' lang='lng.test'  %}
```

1.11.2 Count

Count Module coverts integer of float count to human readable format with the help of `digit` and `bytes` method.

Import Module

```
from pytonik.Functions.count import count
```

Callable

```
count = count()
```

digit returns count of T as trillion, M as Million, K as Thousand.

```
count.digit(432) #this will return 432

count.digit(4324) #this will return 4.3k

count.digit(43242) #this will return 43.2k

count.digit(432427) #this will return 432.4k

count.digit(4324276) #this will return 4.3M
```

bytes returns bytes size of B as Bytes, KB as Kilobyte, MB as Megabyte, GB as Gigabyte and TB as Terabyte

```
count.bytes(444) #this will return B

count.bytes(1024) #this will return 1K

count.bytes(1048576) #this will return 1.00MB

count.bytes(4324273243) #this will return 4.03GB

count.bytes(432434327323243) #this will return 393.30 TB
```

1.11.3 Agent

Agent Module, get users or visitors web browser and operating system information such as name and version

Note: Agent module is a dependency, and it might no longer be maintain under the pytonik repository, if you find this module useful and you want to keep using its features in your project. We recommend installation of `pip install pytonik_agent`.

Import Module

```
from pytonik.Functions.agent import os, browser
```

OS Callable

```
os = os()
```

get operating system name

```
os.name
```

get device

```
os.device
```

Browser Callable

```
browser = browser()
```

get browser name

```
browser.name
```

how get browser version

```
browser.version
```

1.11.4 Ip Address

Ip Module checks visitors/audiences, `proxy`, `sock`, `VPN`, and `IPs` address.

it returns a response such as :- `hostname`, `country`, `city`, `region`, `loc`, `org`

Import Module

```
from pytonik.Functions.ip import ip
```

Callable

```
ip = ip()
```

how to get HTTP IP ADDRESS

```
ip.get().ip
```

how to get VPN IP ADDRESS

```
ip.vpn().ip
```

how to check if visitor is using VPN

To know if visitor is using a `vpn`, we'll need to use `vpn` method with `is_vpn` attribute which will return bool `True` or `False`

```
ip.vpn().is_vpn
```

how to get IP ADDRESS and PROPERTIES

we'll be getting our app visitors `ip`, `hostname`, city, `country` `loc` and `org`

```
visitors = ip.get()
visitors.ip
visitors.hostname
visitors.city
visitors.region
visitors.country
visitors.loc
visitors.org
```

default check IP ADDRESS

Cases whereby there is an `ip` and we want to get the `ip` information we will use `property` method

```
ip.property('41.190.30.100').hostname
ip.property('41.190.30.100').city
ip.property('41.190.30.100').region
ip.property('41.190.30.100').country
ip.property('41.190.30.100').loc
ip.property('41.190.30.100').org
```

1.11.5 Path

Pytonik `path` module handles method `path`, `exist` , `public` argument which return root **path** as string.

Import Module

```
from pytonik.Functions.path import path
```

Callable

```
path = path()
```

Example

```
path.path(path = "users")
```

Example

```
path.public(path)
```

Example

```
path.exist(newpath, defaultpath)
```

1.11.6 Url

Pytonik `url` module handles uniform resource locator notation using `url` method with `path` argument which returns the application link

Import Module

```
from pytonik.Functions.url import url
```

Callable

```
url = url()
```

Example

```
url.url(path = "users")
```

Template Engine

```
{% call url path = "users" %}

{% call url "users"}
```

1.11.7 Readmore

Pytonik `readmore` module helps to hide or limit long content using `lstring` method with the following argument‘‘text‘‘ accept content, empty by default. `trim` accept bool (**True** or **False**) `length` accept integer which is the set limit by default set to **100000000000**. `link` accept string which is the **url** direction, empty by default. `label` accept string, set to Read more by default. `css` accept css(cascading style sheet) as a string and set to **readmore** by default. All default attributes are changable.

Import Module

```
from pytonik.Functions.readmore import readmore
```

Callable

```
readmore = readmore()
```

Example

```
readmore.lstring(text="", trim = 'False', length = '1', link="", label="Read more", ␣
↪css="readmore"):
```

Template Engine

```
{% call readmore text='' trim=True lenght=180 link="url/read/"  %}
```

1.11.8 Iteration

Pytonik iteration module handles `iteration`, enumerate dictionary and `Json`.

Import Module

```
from pytonik.Functions.iteration import iteration
```

Callable

```
iter = iteration()
```

Example Country

```
country = [{ 'country_name': 'Afghanistan'}, {'country_name': 'Aland Islands'}, {
↪'country_name': 'Albania'}']
```

Example Table Result

List Country
Afghanistan
Aland Islands
Albania
Nigeria

Example Country with iteration

```
country = [{ 'country_name': 'Afghanistan'}, {'country_name': 'Aland Islands'}, {
↪'country_name': 'Albania'}']

iter.iteri(country, 'id')
```

Example Iteration Table Result

id	List Country
1	Afghanistan
2	Aland Islands
3	Albania
4	Nigeria

1.11.9 Curl

Pytonik `curl` is an in-built module support sending or initiating actions within or outside pytonik framework. It enables access to API's and return respond back to the application, in form of JSON, HTML, RAW data etc. In this case the use of curl module is to `POST`, `GET`, `HEAD`, `PUT` information in internal or from external API's URL using attributes like `status`, `reason`, and `result`. Whereby `status` handles response codes example **200**, **404**, **500**, etc. which the `reason` of this status could be OK, Not Found, Internal server Error, etc. Get excepted information from `result`

Import Module

```
from pytonik.Functions.curl import curl
```

Callable

```
cl = curl()
```

Curl Local Variable

```
URL #accept url link
HTTPHEADER #httpheader  application/x-www-form-urlencoded etc.
CONTENTHEADER #accept text/plain, html/plain etc.
TIMEOUT #accept
POSTFIELDS #accept dictionary formate {name: example, next: testing}
POST #accept folder or url part / or /mypath
GET #accept folder or url part / or /mypath
HEAD #accept folder or url part / or /mypath
PUT #accept folder or url part / or /mypath
PORT #accept url port 8080
```

GET retrieves information from api's server and returns response `status` , `reason`, and `result`

```
url = "https://example.com"
cl = curl()
cl.set(cl.URL, url)
cl.set(cl.GET, '/users/{username}'.format(username='testme'))
cl.finish()
print(cl.status, cl.reason, cl.result())
```

HEAD check api's and returns response `status` and `reason`

```
url = "https://example.com"
cl = curl()
cl.set(cl.URL, url)
cl.set(cl.HEAD, '/users')
cl.finish()
print(cl.status, cl.reason)
```

POST sent data/information to api using parameters or arguments and returns response `status` , `reason`, and `result`

```
url = "https://example.com"
cl = curl()

cl.set(cl.URL, url)
cl.set(cl.CONTENTHEADER, 'application/x-www-form-urlencoded')
cl.set(cl.ACCEPTHEADER, 'text/plain')
cl.set(cl.POST, '/add/users')
cl.set(cl.POSTFIELDS, {'username':'testme', 'password':'test' })
cl.finish()
print(cl.status, cl.reason, cl.result('utf-8'))
```

HEADER sent data/information to api using parameters or arguments and returns response `status` , `reason`, and `result`

```
url = "https://example.com"
header = {}
cl = curl()
cl.set(cl.URL, url)
cl.set(cl.CONTENTHEADER, 'application/x-www-form-urlencoded')
cl.set(cl.HEADER, header)
cl.finish()
print(cl.status, cl.reason, cl.result('utf-8'))
```

1.11.10 Now

Now module handle time, date functions and accuracy, you might know what time and date are because it happens every date, pytonik provides the best way to handle time date and format with additional future like readable time and date. now module contains methods that support `ago`, `time`, `date`, `datetime`, `create`, `timestamp`, `past`, `future`, `subtract` Now module is usable on both pytonik template engine, controller and model

Import module

```
from pytonik.Functions.now import now
```

callable

```
nowdatetime = now()
```

Ago: covert datetime to readable format `1 year 20 minutes ago` accept string and format as argument `%Y-%m-%d %H:%M:%S`

Example 1 : returns `32 minutes ago.`

```
nowdatetime.ago("2020-01-09 08:32:18")
```

Date: return correct date, let say todays date `2020-01-09` accept `format` as argument, default `format` is set to `%Y-%m-%d`

Example 1 : returns `12:30:59`

```
nowdatetime.date()
```

Time: return correct time, let say my present `12:30:59` accept format as argument, default format is set to `%H:%M:%S`

Example 1: returns `12:30:59`

```
nowdatetime.time()
```

Date: return correct date, let say todays date `2020-01-09 08:18:03` accept format as argument, default format is set to `%Y-%m-%d %H:%M:%S`

Example : returns `2020-01-09 08:18:03`

```
nowdatetime.datetime()
```

Create: This method helps to create new datetime from an existing datetime. In other words changing a previous datetime format to a new datetime format. Let say our present `2020-01-09 08:18:03` and format `%Y-%m-%d %H:%M:%S` we want to change it to `01-09-2020 08:18` and the formation for this will be `%Y-%m-%d %H:%M`.

Example : returns `01-09-2020 08:18`

```
nowdatetime.create("2020-01-09 08:18:03", oldformat="%Y-%m-%d %H:%M:%S", newformat="
↪%Y-%m-%d %H:%M ")
```

Timestamp: return correct unix time and with the same method covert timestamp to date and time. Let say it returns `1578576738` and we want to convert it to datetime. We will need to use the same `timestamp` method and it returns `2020-01-09 08:32:18`

Example 1: returns `1578576738`

```
nowdatetime.timestamp()
```

Example 2: returns `2020-01-09 08:32:18`

```
nowdatetime.timestamp('1578576738')
```

Past: returns previous minutes, hours, days, weeks, seconds, let say we want to go back to 27 days from today date and time.. now in our calendar todays date and time is `2020-01-09 08:32:18`

Example : returns `2019-12-13 08:58:15.983552`

```
nowdatetime.past(days=20)
```

Future: returns Next Date (future) `minutes`, `hours`, `days`, `weeks`, `seconds`, let say we want to look into 27 days from today date and time.. now in our calendar todays date and time is `2020-01-09 08:32:18`

Example: returns `2020-02-05 09:02:08.269823`

```
nowdatetime.future(days=20)
```

Subtract: subtracting or minus a date time from another from date time.. this process comment both date time to provide their format respectively. Argument are `date1`, `format1` and `date2 format2`

Example : returns `27`

```
nowdatetime.subtract(date1='2020-01-09 08:32:18', format1='%Y-%m-%d %H:%M:%S',  date2=
↪'2019-12-13 08:58:15.983552', format2='%Y-%m-%d %H:%M:%S.%f')
```

1.11.11 Extend / Include

Pytonik has a wonderful module that handles both including and extending of external file or paging `include`

and `extend` module helps to structure and manage file architecture. Cases where you have a file named header and all your content or code are saved in it and you want to use it in other file or page of your web application, `include` module handles that purpose but you can `extend` or `include`. At this stage you might be wondering what's difference between the both properties, actually no difference.

This modules are mostly used when working with pytonik Template Engine or html pages

Sample: we are including and extending a file named `header.html` where `home` is the parent folder in our `views` folder and `inc` is a sub folder, using dot `.` sign to separate both folders and file. The last dot signifies last or end of the folder and next is the file. Exception is thrown if your file path or folder cannot be located, this might result in page not found or error path.

Example: Include

```
{% call include 'home.inc.header'  %}
```

Example: Extend

```
{% call extend 'home.inc.header'  %}
```

Let make callable outside template engine

```
from pytonik.Functions.extend import extend
extending = extend()
```

```
extending.extend(path="home.inc.header")
```

1.11.12 Validations

Pytonik `validation` module provides bundles of validity functions that help to validate and trim syntax, string and characters. This Callable module are used when developing application that involves checking of inputted datas or support accuracy in data supply.

Import Module

```
from pytonik.Functions.validation import validation
```

Callable

```
valid = validation()
```

Method `ip` validates only digits and character that contains `.` returns bool `True` or `False` render support IP Address `http://domainname.com`, `https://domainname.com` , `ftp://domainname.com` , `www.domainname.com`

Example

```
url_validations = valid.url('ftp://domainname.com')
```

Method `ip` validates only digits and character that contains `.` returns bool `True` or `False` render support IP Address `0.0.0.0` `123.123.12.1`

Example

```
ipaddress_validation = valid.ip('123.123.12.1')
```

Method `phone` validates only digits and character contains `-` and `+`, returns bool `True` or `False` render support to phone number: `+1-000-000-000`, `10000000000` , `0000000000`, `0000-000-0000`, `00000000000` , `+1000000000`

Example

```
phone_validations = valid.phone("+234-800-000-6000")
```

Method `count` return total count of a string

Example

```
count = valid.count('i love python')
```

Method `email` validates only alphabet and character `.` , `_` , `-` and `@`, returns bool `True` or `False` render support to email address `my_email@gmail.com`, `email@gmail.com` , `my.email@gmail.com`, `my_email@gmail.com`

Example

```
email_validations = valid.email('my_email@gmail.com')
```

Method `fullname` validates full name input field returns `True` or `False` : `firstname lastname prefix firstname lastname`

Example

```
fullname_validations = valid.fullname("prefix firstname lastname")
```

Method `extension` check and validate list of prefix if exist in or as an occurrence in the list, returns `True` or `False`

Example

```
get_extension = valid.extension('filename.jpg', ['png', 'jpg'])
```

Method `length` check and valid the starting length of a string and expected end, where minimum `min` is integer and maximum `max integer ('I love', min, max)` returns `True` or `False`

Example

```
length_validation = valid.length('i love python', 4, 18)
```

1.11.13 Pagination

Pytonik provides pagination module that helps to navigate through pages and tables, it has favorites of methods that meetup expectations `number`, `alphabet` , `alphabet_first_last` , `next_previous`, `first_last` Each of the method has same argument and parameter `total`, `page`, `url`, `css`.

Import Module

```
from pytonik.Functions.pagination import  pagination
```

Callable

```
pagin = pagination()
```

Example: Numbering Pagination

```
pagin.number(total=10, page = 1, url='/blog', css=['pagination', 'page-item', 'page-
↪link'])
```

Example: Alphabet Pagination

```
pagin.alphabet(total=10, page = 'A', url='/blog', css=['pagination', 'page-item',
↪'page-link'])
```

Example: Alphabet First Last Pagination

```
pagin.alphabet_first_last(total=10, page = 'A', url='/blog', css=['pagination', 'page-
↪item', 'page-link'])
```

Example: Next Previous Pagination

```
pagin.next_previous(total=10, page = 1, url='/blog', css=['pagination', 'page-item',
↪'page-link'])
```

Example: First Last Pagination

```
pagin.first_last(total=10, page = 1, url='/blog', css=['pagination', 'page-item',
↪'page-link'])
```

1.12 Schema

pytonik database query schema module provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works on all supported database

systems.

Import Module

```
from pytonik.Driver import Schema
```

Callable

```
DB = Schema.Schema()
```

Tables

```
TABLES = {}

TABLES['users'] = (
        "CREATE TABLE `users` ("
        "  `users_id` int(11) NOT NULL AUTO_INCREMENT,"
        "  `birth_date` date NOT NULL,"
        "  `first_name` varchar(14) NOT NULL,"
        "  `last_name` varchar(16) NOT NULL,"
        "  `email` varchar(255) NOT NULL,"
        "  `votes` int(11) NOT NULL,"
        "  `create_date` date NOT NULL,"
        "  PRIMARY KEY (`emp_no`)"
        ") ENGINE=InnoDB")

TABLES['banks'] = (
        "CREATE TABLE `banks` ("
        "  `banks_id` int(11) NOT NULL AUTO_INCREMENT,"
        "  `users_id` int(11) NOT NULL,"
        "  `bank_name` varchar(40) NOT NULL,"
        "  `bank_sort` varchar(40) NOT NULL,"
        "  PRIMARY KEY (`banks_id`)"
        ") ENGINE=InnoDB")

TABLES['transactions'] = (
        "CREATE TABLE `transactions` ("
        "  `transaction_id` int(11) NOT NULL,"
        "  `users_id` int(11) NOT NULL,"
        "  `amount` varchar(40) NOT NULL,"
        "  `from_date` date NOT NULL,"
        "  `to_date` date NOT NULL,"
        "  PRIMARY KEY (`transaction_id`,`from_date`), KEY `transaction_id`␣
↪(`transaction_id`),"
        ") ENGINE=InnoDB")
```

1.12.1 create

Create table using `create` method

```
DB.table(TABLES).create()
```

1.12.2 drop

If you wish to `drop` the entire table, you may use the `drop` method:

Drop All

```
DB.table('users').drop()
```

Drop if Exist

```
DB.table('users').exists().drop()
```

1.12.3 insert

The query schema also provides an insert method for inserting records into the database table. The `insert` method accepts an array of column names and values:

```
insert = DB.table('users').insert(
                [
                dict(email ='info@pytonik.com', name= 'Pytonik MVC', created_at='2020-
↪02-05 09:02:08.26'),
                ])
```

Multiple Insert

You may even insert several records into the table with a single call to insert by passing an array of arrays. Each `dictionary` inside `list` represents a row to be inserted into the table:

```
insert = DB.table('users').insertGetId(
                [
                dict(email ='dev@pytonik.com', name = 'Pytonik Moduel', created_at=
↪'2020-02-05 09:02:08.26'),
                ])
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the insertGetId method to insert a record and then retrieve the ID:

```
insert = DB.table('users').insert(
                [
                dict(email ='info@pytonik.com', name= 'Pytonik MVC', created_at='2020-
↪02-05 09:02:08.26'),
                dict(email ='dev@pytonik.com', name = 'Pytonik Moduel', created_at=
↪'2020-02-05 09:02:08.26'),
                ])
```

Note: When using PostgreSQL the insertGetId method expects the auto-incrementing column to be named id. If you would like to retrieve the ID from a different `sequence`, you may pass the column name as the second parameter to the insertGetId method.

1.12.4 update

In addition to inserting records into the database, the query schema can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an `dict` of column and value pairs containing the columns to be updated. You may constrain the update query using where clauses:

Update

```
DB.table('users').where('id', '=', 18).update([dict(email='info@pytonik.com')]))
```

1.12.5 delete

The query schema may also be used to `delete` records from the table via the delete method. You may constrain delete statements by adding where clauses before calling the delete method:

```
DB.table('users').delete()
```

```
DB.table('users').where('users_id', 1).delete()
```

1.12.6 selects

If you don't even need an entire row, you may extract a single value from a record using the value method. This method will return the value of the column directly:

```
DB.table('users').value('users_username', 'email').select().get()
```

```
DB.table('users').where('users_id', 18).select().get()
```

The query schema also provides a variety of aggregate methods such as `counts`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

MAX()

```
DB.table('transactions').max('amount').select().get()
```

MIN()

```
DB.table('transactions').min('amount').select().get()
```

AVG

```
DB.table('transactions').avg('amount').select().get()
```

COUNT()

```
DB.table('transactions').counts().select().get().result
```

You may combine these methods with other methods:

```
DB.table('transactions').where('status', 1).min('amount').select().get()
```

Determining If Records Exist

Instead of using the count method to determine if any records exist that match your query's constraints, you may use the exists and `notExist` methods:

Example: **Exist**

```
DB.table('orders').where('finalized', 1).exists()
```

Example: **notExist**

```
DB.table('orders')->where('finalized', 1).notExist()
```

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from the database table, you may use the first method. This method will return a single dictionary object `{}`:

```
user = DB::table('users')->where('status', 1)->first()

print(user["name"])
```

Retrieving A List Of Column Values

If you would like to retrieve a Collection containing the values of a single column, you may use the pluck method. In this example, we'll retrieve a Collection of role titles:

```
titles = DB::table('roles').pluck('title');

for  title in titles:
     print(title)
```

You may also specify a custom key column for the returned Collection:

```
roles = DB.table('roles').pluck('title', 'name');

for title in roles:

    print(title["name"])
```

If you need to work with thousands of database records, consider using the `chunk` method. This method retrieves a small chunk of the results at a time and feeds each chunk into a Closure for processing. This method is very useful for writing Artisan commands that process thousands of records. For example, let's work with the entire users table in chunks of 100 records at a time:

```
DB.table('users').orderBy('id').chunk(100)
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. So, when updating records while chunking, it is always best to use the chunkById method instead. This method will automatically paginate the results based on the record's primary key:

```
users = DB.table('users').where('status ', 'PENDING').orderBy('users_id').chunk(100,
DB.table('countries').where('users_id', '{users_id}').updates([dict(vote='200',␣
↪create_at='20/01/2020')])
    )
```

Note: When updating or deleting records inside the chunk callback, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the chunked results.

Select Value

If you don't even need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly

Example 1.0:

```
DB.table('users').select('users_username', 'email').get()
```

Example 1.1:

```
DB.table('users').value('users_username', 'email').select().get()
```

Select Where with custom column

Example 1.0:

```
DB.table('users').where('users_id', '=', 1).select('users_username','users_email').
↪get()
```

Example 1.1:

```
DB.table('users').where('users_id', '=', 1).value('users_username','users_email').
↪select().get()
```

Select groupBy

```
DB.table('users').groupBy('users_id').select().get()
```

Select groupBy/having

The `groupBy` and `having` methods may be used to group the query results. The having method's signature is similar to that of the where method:

```
DB.table('users').groupBy('users_id').having('permission', '>', '100').select().get()
```

You may pass multiple arguments to the `groupBy` method to group by multiple columns:

```
DB.table('users')->groupBy('first_name', 'status')->having('permission', '>', '100').
↪select().get()
```

orderBy

The `orderBy` method allows you to sort the result of the query by a given column. The first argument to the `orderBy` method should be the column you wish to sort by, while the second argument controls the direction of the sort and may be either `asc` or `desc`:

```
DB.table('users').orderBy('users_id', 'desc').select().get()
```

```
DB.table('users').groupBy('users_id').orderBy('users_id', 'desc').select().get()
```

limit

To limit the number of results returned from the query, or to `skip` a given number of results in the query, you may use the `skip` and `take` methods:

Example 1.0:

```
DB.table('users').skip(1).take(2).select().get()
```

limit with offset

Example 1.1:

```
DB.table('users').offset(1).limit(2).select().get()
```

Alternatively, you may use the limit and offset methods:

```
DB.table('users').limit(1).select().get()
```

Select limit with offset

Example 1.1:

```
DB.table('users').offset(1).limit(2).select().get()
```

1.12.7 where

Where having

```
DB.table('users').where('status', 1).having('permission', '>', 2).select().get()
```

You may use the `where` method on a query schema instance to add `where` clauses to the query.

The most basic call to `where` requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. Finally, the third argument is the value to evaluate against the column.

For example, here is a query that verifies the value of the "votes" column is equal to 100:

```
DB.table('users').where('votes', '=', 1).select().get()
```

For convenience, if you want to verify that a column is equal to a given value, you may pass the value directly as the second argument to the where method:

```
DB.table('users').where('votes', 1).select().get()
```

You may use a variety of other operators when writing a `where` clause:

```
DB.table('users').where('votes', '>=', 100).select().get()

DB.table('users').where('votes', '<>', 100).select().get()

DB.table('users').where('votes', 'like', 'T%').select().get()
```

Or Statements

You may chain where constraints together as well as add or clauses to the query. The `orWhere` method accepts the same arguments as the where method:

```
DB.table('users').where('user_id', 15).orWhere('user_id', 15).select('email')
```

AND

```
DB.table('users').where('user_id', 18).where('email', 'info@pytonik.com').select().
↪get()
```

OR

```
DB.table('users').where('user_id', 18).orWhere('user_id', 15).select().get()
```

AND/OR

```
DB.table('users').where('user_id', 18).where('email', 'info@pytonik.com').orWhere(
↪'user_id', 15).select().get()
```

Where Column

The `whereColumn` method may be used to verify that two columns are equal:

```
DB.table('users').whereColumn('first_name', '>', 'last_name')
```

Multiple Where Column

```
DB.table('users').whereColumn(('first_name', '=', 'last_name'),('updated_at', '>',
↪'created_at'))
```

Additional Where Clauses

Where Between

The `whereBetween` method verifies that a column's value is between two values:

```
DB.table('transactions').whereBetween('votes', ['30']).select()
```

Where Between

The `whereNotBetween` method verifies that a column's value lies outside of two values:

```
DB.table('transactions').whereNotBetween('votes', ['1', '100', '30']).select()
```

whereIn / whereNotIn

The `whereIn` method verifies that a given column's value is contained within the given list:

```
DB.table('users').whereIn('id', [1, 2, 3]).select().get()
```

The `whereNotIn` method verifies that the given column's value is **not** contained in the given list:

```
DB.table('users').whereNotIn('id', [1, 2, 3]).select().get()
```

whereNull / whereNotNull

The `whereNull` method verifies that the value of the given column is `NULL`:

```
DB.table('users').whereNull('updated_at').select().get()
```

The `whereNotNull` method verifies that the column's value is not `NULL`:

```
DB.table('users').whereNotNull('updated_at').select().get()
```

The query schema also provides a quick way to `union` two queries together. For example, you may create an initial query and use the `union` method to `union` it with a second query:

```
first = DB.table('username').select().set()

users = DB.table('countries').orderBy('country_id').select().union(first).get()
```

Note: The `unionAll` method is also available and has the same method signature as `union`.

1.12.8 join

The query schema may also be used to write join statements. To perform a basic `inner join`, you may use the `join` method on a query schema instance. The first argument passed to the join method is the name of the table you need to `join` to, while the remaining arguments specify the column constraints for the join. You can even join to multiple tables in a single query:

Join

```
DB.table('users').join('contacts', 'users.id', '=', 'contacts.user_id').select().get()
```

Inner Join

```
DB.table('users').join('contacts', 'users.id', '=', 'contacts.user_id').join('orders',
↪ 'users.id', '=', 'orders.user_id').select('users.*', 'contacts.phone', 'orders.
↪price').get()
```

1.12.9 Left Join / Right Join

If you would like to perform a "left join" or "right join" instead of an "inner join", use the `leftJoin` or `rightJoin` methods. These methods have the same signature as the `join` method:

left Join

```
DB.table('users').leftJoin('bank', 'bank.users_id, '=', 'users.users_id').select().
↪get()
```

left Join with Join

```
DB.table('users').leftJoin('bank', 'bank.users_id, '=', 'users.users_id').join(
↪'message', 'message.users_id', '=', 'users.users_id').select().get()
```

right Join

```
DB.table('users').rightJoin('bank', 'bank.users_id', '=', 'users.users_id').select().
↪get()
```

right Join with

```
DB.table('users').rightJoin('bank', 'bank.users_id', '=', 'users.users_id').join(
↪'message', 'message.users_id', '=', 'users.users_id').select().get()
```

left outer Join

```
DB.table('comment').where('comment_status', '=', 1).fromTable('comment').outerJoin(DB.
↪raw("(SELECT parent_id, COUNT(*) AS comment FROM parent GROUP BY parent_id) as sub
↪"), 'comment_id', 'sub.parent_id ').select('a.comment_id', 'a.comment_name', 'a.
↪comment_status', 'a.comment_link', 'sub.Count')
```

Advanced Join

You may also specify more advanced join. To get started, pass a Closure as the second argument into the `join` method. The Closure will receive a JoinClause object which allows you to specify constraints on the join clause:

```
DB.table('users').join('bank', 'bank.users_id', '=', 'users.users_id').where('bank.
↪status', '>', 5).select().get()
```

If you would like to use a "where" style clause on your joins, you may use the `where` and `orWhere` methods on a join. Instead of comparing two columns, these methods will compare the column against a value:

1.12.10 Raw Expressions

Sometimes you may need to use a raw expression in a query. To create a raw expression, you may use the `DB.raw` method:

DB.raw

```
DB.table('users').where('status', '>', 1).groupBy('status').select(DB.raw('count(*)␣
↪as user_count, status')).get()
```

1.13 Advance

We have lot of features to look into if we really want to achieve more using **pytonik**, I will choose to be a web developer and I want to be very good in web development, but before I can become more better, I need to know how to go about using all the modules provided by **pytonik**. Knowing how to used will not make me better but limited. I think I need to improved and break my limit but now how can I break my limit. The best way to do so, is to know how to create my own custom `model` and `controller`. Great, I believe I can now start developing scalable web application.

1.13.1 Model

Model, is breaking application into parts, in which each part as it model and can as well relate to another part. Let say we have a model that is called `Users`, and `Result`, these models are closely related, each model cannot do without other. We will like to talk more about how to work with model but making a real live example will explain better. Model are created and saved into `model` folder. We will need to import Model and inherit Model properties into our newly created model. below example will explained better.

Example: Users.py

```
from pytonik.Model import Model

class Users(Model):

        def __getattr__(self, item):
                return item

        def __init__(self):
                return None

        def get(self, userid=""):
                returns "user"

        def list(self):
                return "all user"

        def delete(self, id=""):
                if id is not "":
                        return "delete user with id"
                else:
                        return "delete  all user"
```

Example 1.0: Result.py

```
from pytonik.Model import Model

class Result(Model):

        def __getattr__(self, item):
                return item

        def __init__(self):
                return None

        def get(self, userid=""):
                return "user result"

        def list(self):
                return "all result"

        def deletebyuser(self, userid="", id=""):
                if userid is not "" and id is not "":
                        return "delete user result that as id"
                else:
                        return "delete user result"

        def delete(self, id=""):
                if id is not "":
                        return "delete  result with id"
                else:
                        return "delete  all result"
```

I will like to know more, at this stage I found the example very interesting, now that I have learnt how to create model, feel I am still missing out, because I need to learn how to implement database query. `Pytonik` has an hand build **schema** that will handle all database features. In our case we will use **Schema** `method` and `attribute`, this module was design to enhance database structuring. Since we are using Model Module we should be comfortable using database properties without `calling` or `importing` another module. if you have not learn more about **pytonik** `Schema`, I prefer you should to read more about how to use it because you will need it in the future if not now.

Example 1.1: Result.py

```
from pytonik.Model import Model

class Result(Model):

   def __getattr__(self, item):
        return item

   def __init__(self):
        self.result = self.table('result')
        return None

   def get(self, userid=""):
       query= self.result.where('users_id','=',userid).select().get()
       return query.rowCount, query.result

   def list(self):
       query= self.result.select().get()
       return query.rowCount, query.result
```

(continues on next page)

(continued from previous page)

```
  def deletbyuser(self, userid="", id=""):
        if userid is not " and id is not "":
            query = self.result.where('users_id','=',userid).and ('result_id' '=',␣
↪id).delete()
            return query
        else:
            query = self.result.where('users_id','=',userid).delete()
            return query

  def delete(self, id=""):
        if id is not "":
            query = self.result.where('result_id','=',id).delete()
            return query
        else:
            query = self.result.delete()
            return query
```

1.13.2 Controller

Controller is the heart of the application, it is the most important part of application and can function without the help of model, but the `model` cannot function without `controller`. The controller handles the result and send action in and out of the application. Controller controls and send data to the browser using the help of `view` which is a method in `App` module. All controller files are stored in `controller` folder and are saved/stored in Capitalized form example `UsersController.py`. if file is saved `userscontroller.py` or `Userscontroller.py` are not accepted and will definitely lead to exception.

Example: UsersController.py

The illustration shows how to create controller and implement `views` module, which is one of the property of `App` module, as you can see we are sending data `user.html` which is stored in our `views` folder in our **project directory**.

```
from pytonik.App import App

mvc = App()

def index():

data = {
'title': "pytonik MVC",
'label': "List Pytonik Users"

}
mvc.views('user', data)
```

Example: user.html

Here we can see that we are can display variable in `user.html` sent from `UsersController.py`

```
<html>

<head>
<title>{{title}}</title>
</head>
```

(continues on next page)

(continued from previous page)

```
<body>

    <h1>{{label}}</h1>

</body>

</html>
```

How to load ‘‘model‘‘ in ‘‘controller‘‘

Example 1.0: load model `Users.py` into controller `UsersController.py`

once `Users` model load into `UsersController` it gives Controller access to all the methods and attribute in Users model. we can call each of the method defined in `Users`

```
from pytonik.App import App
from pytonik.Model import Model

mvc = App()
users = Model.Load('Users')

def index():

data = {
'title': "pytonik MVC",
'label': "List Pytonik Users”

}
mvc.views('user', data)
```

Note: if we keep importing module each time we want to make use of them, then we will write a huge lines of codes which is not what we want. pytonik has a module called `Web`, it gives access to bunch of modules, so we will not have to be importing module into our controller. below example will explain.

Example 1.1: load model `Users.py` into controller `UsersController.py` . `load` is a method in `Model` module

```
from pytonik import Web

mvc = Web.App()
users = Web.Load('Users')

def index():

data = {
'title': "pytonik MVC",
'label': "List Pytonik Users”

}
mvc.views('user', data)
```

Note: `App` Module has three important methods `header` is called when displaying strings or characters, `redirect` from initial page to the preferred page. `referer` from initial to the previous page.

`header` method has `type` argument with default value `text/html`.

Example:

```
from pytonik import Web
mvc = Web.App()

def index():

    mvc.header()

    print("i love pytonik")
```

`redirect` method has `location` argument with default value `/`.

Example 1.0:

```
from pytonik import Web
mvc = Web.App()

def index():
    mvc.redirect('/login')
```

Example 1.1: Using `url` function together with `redirect` method

```
from pytonik import Web
mvc = Web.App()

def index():

    mvc.redirect(Web.url('/login'))
```

`referer` method has `location` argument with default value `/`.

Example 1.0:

```
from pytonik import Web
mvc = Web.App()

def index():
    mvc.referer()
```

Note: Cases where referer page does not exist, set an alternative location `referer('home')`. Let say the previous page is not found, we have to provide an alternative location. This means we are directing to **home** page

Example 1.1: Using `url` function together with `referer` method

```
from pytonik import Web
mvc = Web.App()

def index():

    mvc.referer(Web.url('/home'))
```