pycommunicate Documentation

Release 0.0.7

mincrmatt12

June 12, 2016

1 What is pycommunicate? 1

2 Example 3
2.1 Getting up and running . . . 4
2.2 How pycommunicate works . . . 4
2.3 Creating your first pycommunicate app . . . 5
2.4 Sprinkling in some server-side stuff . . . 7
2.5 CommunicateApp . . . 13
2.6 Controller and ControllerFactory . . . 14

Python Module Index 17

What is pycommunicate?

pycommunicate is a python web-server library designed to make server-heavy webapps easy!

In other words, this library is similar to others (in fact it wraps flask to work), but with one very amazing difference: **you can modify the dom from the server**

You can attach events to elements, change properties, and much more!

Want to get started? Well you're in the right place, because this *is* the documentation! :)

Example

This is code from examples/hello.py found in the docs folder.

```
"""
hello.py:

Using the pycommunicate framework to serve a static webpage, as well as show internal id numbers. It
session variables.

Because the returned value does not load the js libs, the socketio connection is not started. This me
none of the apis will work nor will load ever be called. Again, this is a very simple demo.

WARNING: DO NOT ACTUALLY SHOW THE USER ID TO A USER!!
"""

from pycommunicate.server.app.communicate import CommunicateApp
from pycommunicate.server.bases.controller import ControllerFactory
from pycommunicate.server.bases.views import View


class HelloView(View):
    def render(self):
        if 'count' in self.controller.user.session:
            self.controller.user.session['count'] += 1
        else:
            self.controller.user.session['count'] = 1
        return ("Hello World! My user id is {}, and my request id is {}. The session says you have re
                " times").format(
            self.controller.user.id_,
            self.controller.user.request_id,
            self.controller.user.session['count'])


app = CommunicateApp()
controller = ControllerFactory().add_view(HelloView).set_default_view(HelloView)

app.add_controller("/", controller)

app.set_secret_key("secret!!!")
app.run()
```

2.1 Getting up and running

Contents

- *Getting up and running*
  - *Installing pycommunicate*

2.1.1 Installing pycommunicate

Alright, first things first: *get the library*

If you don't have pip, go get it using get-pip.py.

Then, in a correctly elevated terminal run:

```
pip install pycommunicate
```

If you run windows, you may need to use:

```
python -m pip install pycommunicate
```

Both will do the same thing.

That's it! You have now installed pycommunicate and are ready to proceed to the next step!

2.2 How pycommunicate works

Before we can start writing code, let's get some terms and stuff out of the way:

controller A collection of views that manages one specific route. Can be thought of as a 'page'

view One page option for a controller, the bulk of ui code and the render function resides here. Can be thought of as a 'sub-page'

request A unique request for a controller.

app An instance of CommunicateApp. This is the main thing inside pycommunicate based webapps.

Now lets explain them in a bit more detail:

2.2.1 Controller

A controller is probably not what you think it is, as this does not follow MVC. Instead, controllers are used to hold many views, which actually deal with the page. Controller do, however, contain the controller session, which is one of many sessions in pycommunicate. This one remains across any given request.

Controllers are not created by you, though. They are created by pycommunicate itself, you only define one. For this you use ControllerFactory. You can subclass both Controller and ControllerFactory to create your own custom behaviour, however.

2.2.2 View

Views probably contain the bulk of your code. They are responsible for serving up a page, handling events in a page, and much more. Because of this, views are subclassed from the base View class.

OK, I promise in the next part we can start actually programming something!

2.3 Creating your first pycommunicate app

Alright, now we can actually start.

The tutorial project will be quite simple:

- will serve up a todo-file
- will allow adding and removing of entries to it

As you can see, this will be very simple, but using other libraries might require lots of ajax processing. In pycommunicate, none of this is needed!

Let's get started by making the simplest example: serving up a static page.

2.3.1 Setting up our app

Let's start by making an empty directory called tutorial somewhere on your system. Inside this directory should be one file and one folder: main.py and templates. We'll get back to templates in a second, but for now let's make main.py.

2.3.2 Making our page

First of all, let's create a simple page for our app in html. Some of it might not make sense right now, but just roll with it. For now just put this into templates/home.html:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Tutorial Checklist</title>
    {{ add_includes() }}
</head>
<body>
<h1>Welcome to the TODO!</h1>
<div id="todo">
    <p id="loadingBar">Please wait... loading todos</p>
</div>
<br />
<input type="text" id="next" /><button id="add">Add</button>
</body>
</html>
```

2.3.3 Serving up the page statically

First of all, we will need a *pycommunicate.server.app.communicate.CommunicateApp* instance before we even start writing anything, so lets add that now to main.py, and while we're at it, lets import everything we'll need:

```python
import eventlet
from pycommunicate.server.bases.views import View
from pycommunicate.server.bases.controller import ControllerFactory
from pycommunicate.server.app.communicate import CommunicateApp

app = CommunicateApp()  # main app instance
```

Now, lets create a view called TodoView, and have it display the content inside home.html.

```python
class TodoView(View):
    def render(self):
        return self.controller.templater.render("home.html")
```

Alright, let's break this down:

```python
class TodoView(View):
```

This defines our view, as all views are subclasses of View, from pycommunicate.server.bases.views

Next, we have the render() method. This is called to get the base page to serve when requested. It is the only thing you actually have to override. Views have references to their parent controllers, which have Templaters. A `pycommunicate.templating.Templater` will render a template, the default location is templates/, but this can be changed. See *`pycommunicate.server.app.communicate.CommunicateApp`* for how to change it.

Note: The templater, and all of pycommunicate use jinja2, a templating engine. For more info on what we provide in jinja2, and how to use the templater, go look at its page.

Anyways, now that we have a view, we should create a controller:

```python
controller = ControllerFactory().add_view(TodoView).set_default_view(TodoView)
app.add_controller('/', controller)
```

This one is pretty simple, we create a `ControllerFactory` and call its methods (which are chainable) to add the TodoView view, and set it as the default (initial) one.

We're almost done, now, and all we have to add is the secret key and the call to run.

Danger: Remember, the secret key must be kept secret. For a truly random secret key, use os.urandom()

```python
app.set_secret_key('secret!')
app.run()
```

And that's it! You should be able to simply run it and see a static, lifeless page. On the next page, we'll get to adding some event handlers. Here's the full source right now for copying:

```python
import eventlet

from pycommunicate.server.bases.views import View
from pycommunicate.server.bases.controller import ControllerFactory
from pycommunicate.server.app.communicate import CommunicateApp

app = CommunicateApp()


class TodoView(View):
    def render(self):
        return self.controller.templater.render("home.html")

controller = ControllerFactory().add_view(TodoView).set_default_view(TodoView)
app.add_controller("/", controller)

app.set_secret_key("secret!")
app.run()

```

2.4 Sprinkling in some server-side stuff

Alright, now that we've got a page working, let's add some functionality.

Before we do that, though, add this class to the main.py file. It contains some multithreading and fileio stuff to make it easy to add and remove tasks.

```
class TodoReader:
    def __init__(self):
        self.todos = {}
        self.last_todo = 0
        try:
            with open('todo.txt', 'r') as old:
                for i, line in enumerate(old.readlines()):
                    self.todos[i] = line
                    self.last_todo = max(self.last_todo, i)
        except IOError:
            pass
        self.add_queue = eventlet.queue.Queue()
        self.adds = {}
        self.remove_queue = eventlet.queue.Queue()

    def add_daemon(self):
        while True:
            add = self.add_queue.get()

            self.todos[self.last_todo + 1] = add
            self.adds[add].put(self.last_todo + 1)
            self.last_todo += 1

            with open('todo.txt', 'w') as new:
                for index in self.todos:
                    new.write(self.todos[index] + ("\n" if not self.todos[index].endswith("\n") else

    def del_daemon(self):
        while True:
            d = self.remove_queue.get()

            del self.todos[d]
            self.last_todo = 0
            for i in self.todos:
                self.last_todo = max(self.last_todo, i)

            with open('todo.txt', 'w') as new:
                for index in self.todos:
                    new.write(self.todos[index] + ("\n" if not self.todos[index].endswith("\n") else

    def start(self):
        pool = eventlet.greenpool.GreenPool(2)
        pool.spawn_n(self.add_daemon)
        pool.spawn_n(self.del_daemon)

    def add(self, text):
        self.add_queue.put(text)
        self.adds[text] = eventlet.queue.Queue()

    def wait_on(self, text):
        ind = self.adds[text].get()
        del self.adds[text]
        return ind

    def remove(self, index):
        self.remove_queue.put(index)
```

This should go somewhere after the creation of `app` instance

After that, but before the view code, add this to properly initialize it:

```
todo = TodoReader()
todo.start()
```

Alright, so now we have a variable called `todo` that manages... todos!

2.4.1 Showing the current todos and removing them

At the heart of server-side DOM manipulation in pycommunicate is `HTMLWrapper`. For all of the methods it supports, go look at it, but the one we will be using is `element_by_selector()`. This method will return a `ElementWrapper` tracked to follow the selector given. This can then be used to modify the DOM.

Add a load() method

When a user loads a page with the pycommunicate libraries installed, as soon as `document.ready()` occurs client-side, the library will connect to the server and when the server finishes initializing the connection, the active view's `load()` method is called.

Let's create this method and add some code to it after the `render()` method:

```
def load(self):
    todo_div = self.html_wrapper.element_by_selector("#todo")

    loading_message = self.html_wrapper.element_by_selector("#loadingBar")
    loading_message.delete()

    for index in todo.todos:
        text = todo.todos[index]
        todo_page_div = todo_div.append_element_inside_self("div", "todo{}".format(index))
        text_p = todo_page_div.append_element_inside_self("p", "todoText{}".format(index))
        text_p.content.set(text)
        button = todo_page_div.append_element_inside_self("button", "todoRemove{}".format(index))
        button.add_event_listener("click", self.make_handler(index))
        button.content.set("Remove")
```

After doing this, if you are using an IDE, it will complain about self.make_handler not existing, so make that too:

```python
def make_handler(self, index):
    def handler():
        todo.remove(index)

        todo_div = self.html_wrapper.element_by_selector("#todo{}".format(index))
        todo_div.delete()
    return handler
```

Alright, so lets explain what's actually happening there

What's going on in the load() function ?!

Alright, let's do this bit by bit:

```python
todo_div = self.html_wrapper.element_by_selector("#todo")
```

The first part is probably self-explanatory to all python programmers, so let's explain that call. As I said earlier the `element_by_selector()` method returns a `ElementWrapper`. In this case, it is tracking the first thing with an id of `todo`. In our html file, that points to the `<div>`. So this call will set todo_div equal to something that represents... the todo div!

```python
loading_message = self.html_wrapper.element_by_selector("#loadingBar")
loading_message.delete()
```

The first line does similar things to the example above, so lets explain the second line. It calls `loading_message`'s `delete()` method, which deletes the element. This effectively clears the "Loading..." message from the page.

```python
for index in todo.todos:
    text = todo.todos[index]
    todo_page_div = todo_div.append_element_inside_self("div", "todo{}".format(index))
    text_p = todo_page_div.append_element_inside_self("p", "todoText{}".format(index))
    text_p.content.set(text)
```

So the loop goes through every todo in the `TodoReader`. This class uses ids and a dictionary to store todos, so we loop through the keys, which are the indices.

Note: Although I could of used a list, this seemed easier to implement and keep track for removing entries, so I used a dictionary.

For each todo, get its text and store it in `text`. Then, use the element creation function `append_element_inside_self()` to create and get a `<div>` element with id `todo{index}` inside the `todo_div`. The next call is very similar, only calling it on `todo_page_div` and using it to create a `<p>` element with id `todoText{index}` instead.

The `content` attribute is a wrapper for `innerText`, which can be used to get or set the text of an element. We use this to change the text of the new `<p>` element to the text of the todo.

Warning: The get functions of element properties block until the property is received, while the set() functions return as soon as the change is submitted to be sent. This means that calls to set() and then immediately after get() can return the wrong values. This will probably be changed in a later version, or an option added to block on the set() call.

```python
button = todo_page_div.append_element_inside_self("button", "todoRemove{}".format(index))
button.add_event_listener("click", self.make_handler(index))
button.content.set("Remove")
```

Line 12 and 14 use already explained functions, so I'll detail the `add_event_listener()` method instead. This method will attach an event to a js event. These are using the chrome and firefox names, not the IE ones. We use it here to attach the button's click method to a dynamically generated event handler setup to destroy the todo server-side, and then use `ElementWrapper.delete()` to remove it from the client-side.

2.4.2 Adding todos

Add code to the load() method

To do this, you need to add the following lines at the end of `load()`:

```
add_button = self.html_wrapper.element_by_selector("#add")
add_button.add_event_listener("click", self.add_handler)
```

I've already explained above what this does, so let's go create that add_handler event handler.

The add_handler method

The add_handler method will deal with when the user clicks the "Add" button. Here's the code in it:

```
1  text = "- " + self.html_wrapper.element_by_selector("#next").get_property("value")
2  todo.add(text)
3  self.html_wrapper.element_by_selector("#next").set_property("value", "")
4  index = todo.wait_on(text)
5  todo_div = self.html_wrapper.element_by_selector("#todo")
6  todo_page_div = todo_div.append_element_inside_self("div", "todo{}".format(index))
7  text_p = todo_page_div.append_element_inside_self("p", "todoText{}".format(index))
8  text_p.content.set(text)
9  button = todo_page_div.append_element_inside_self("button", "todoRemove{}".format(index))
10 button.add_event_listener("click", self.make_handler(index))
11 button.content.set("Remove")
```

Lines 5-11 are simply copied from the `load()` function, so look there for info on what these do.

Again, I'll go line by line.

```
text = "- " + self.html_wrapper.element_by_selector("#next").get_property("value")
```

This will set text to "- " plus whatever is in the input field. The `get_property()` method will return whatever the JS element defined by the `ElementWrapper` has for that name. The `value` property contains the content of the input field.

Line 3 simply empties it using `set_property()`.

Lines 2 and 4 use the `TodoReader` to add and retrieve the index for the new entry, and the rest is just as above.

2.4.3 Putting it all together

Your main.py file should now look like this:

```
import eventlet

from pycommunicate.server.bases.views import View
from pycommunicate.server.bases.controller import ControllerFactory
from pycommunicate.server.app.communicate import CommunicateApp

```

```
app = CommunicateApp()

# This class uses some greenlet things and is beyond the scope of this tutorial


class TodoReader:
    def __init__(self):
        self.todos = {}
        self.last_todo = 0
        try:
            with open('todo.txt', 'r') as old:
                for i, line in enumerate(old.readlines()):
                    self.todos[i] = line
                    self.last_todo = max(self.last_todo, i)
        except IOError:
            pass
        self.add_queue = eventlet.queue.Queue()
        self.adds = {}
        self.remove_queue = eventlet.queue.Queue()

    def add_daemon(self):
        while True:
            add = self.add_queue.get()

            self.todos[self.last_todo + 1] = add
            self.adds[add].put(self.last_todo + 1)
            self.last_todo += 1

            with open('todo.txt', 'w') as new:
                for index in self.todos:
                    new.write(self.todos[index] + ("\n" if not self.todos[index].endswith("\n") else

    def del_daemon(self):
        while True:
            d = self.remove_queue.get()

            del self.todos[d]
            self.last_todo = 0
            for i in self.todos:
                self.last_todo = max(self.last_todo, i)

            with open('todo.txt', 'w') as new:
                for index in self.todos:
                    new.write(self.todos[index] + ("\n" if not self.todos[index].endswith("\n") else

    def start(self):
        pool = eventlet.greenpool.GreenPool(2)
        pool.spawn_n(self.add_daemon)
        pool.spawn_n(self.del_daemon)

    def add(self, text):
        self.add_queue.put(text)
        self.adds[text] = eventlet.queue.Queue()

    def wait_on(self, text):
        ind = self.adds[text].get()
        del self.adds[text]
        return ind
```

```python

    def remove(self, index):
        self.remove_queue.put(index)

todo = TodoReader()
todo.start()  # start up the TodoReader.


class TodoView(View):
    def render(self):
        return self.controller.templater.render("home.html")

    def make_handler(self, index):
        def handler():
            todo.remove(index)

            todo_div = self.html_wrapper.element_by_selector("#todo{}".format(index))
            todo_div.delete()
        return handler

    def add_handler(self):
        text = "- " + self.html_wrapper.element_by_selector("#next").get_property("value")
        todo.add(text)
        self.html_wrapper.element_by_selector("#next").set_property("value", "")
        index = todo.wait_on(text)
        todo_div = self.html_wrapper.element_by_selector("#todo")
        todo_page_div = todo_div.append_element_inside_self("div", "todo{}".format(index))
        text_p = todo_page_div.append_element_inside_self("p", "todoText{}".format(index))
        text_p.content.set(text)
        button = todo_page_div.append_element_inside_self("button", "todoRemove{}".format(index))
        button.add_event_listener("click", self.make_handler(index))
        button.content.set("Remove")

    def load(self):
        # add existing todos:
        todo_div = self.html_wrapper.element_by_selector("#todo")

        loading_message = self.html_wrapper.element_by_selector("#loadingBar")
        loading_message.delete()

        for index in todo.todos:
            text = todo.todos[index]
            todo_page_div = todo_div.append_element_inside_self("div", "todo{}".format(index))
            text_p = todo_page_div.append_element_inside_self("p", "todoText{}".format(index))
            text_p.content.set(text)
            button = todo_page_div.append_element_inside_self("button", "todoRemove{}".format(index)
            button.add_event_listener("click", self.make_handler(index))
            button.content.set("Remove")

        add_button = self.html_wrapper.element_by_selector("#add")
        add_button.add_event_listener("click", self.add_handler)

controller = ControllerFactory().add_view(TodoView).set_default_view(TodoView)
app.add_controller("/", controller)

app.set_secret_key("todo_secrets!")
app.run()

```

If it looks like that (give or take some whitespace or comments) then you're good to go! Simply run it and connect to it with the link in the console and watch your creation work!!

This is the end of the tutorial, but I'm sure you could do other stuff with this if you want.

2.5 CommunicateApp

class `pycommunicate.server.app.communicate.`**`CommunicateApp`**(*self*, *web_port=8080*, *host='localhost'*, *debug=False*, *maximum_handler_threads=10000*, *template_directory="templates"*)

This is the main app instance, used to create and run a pycommunicate server. Its arguments are for configuration

Parameters

- **`web_port`** (*`int`*) – The port to host the server on
- **`host`** (*`str`*) – The hostname to host on
- **`debug`** (*`bool`*) – Run with the debug server, never user in production
- **`maximum_handler_threads`** (*`int`*) – The size of the handler event pool. This may be removed in the future
- **`template_directory`** (*`str`*) – Relative path to the templates

`set_secret_key`(*key*)

Sets the internal secret key

Danger: The secret key must be kept secret, for a truly random key use `os.urandom()`

Parameters **`key`** (*`str`*) – The new secret key

`add_controller`(*route*, *controller*)

Associates a *`ControllerFactory`* with a route.

Warning: Any route starting with `__pycommunicate/` will not work, as this path is reserved for pycommunicate's internal routes.

Tip: Routes can contain variable parts, indicated like this: `<name>` where name is some name. You can also use integers, for that use `<int:name>`.

Parameters

- **`route`** (*`str`*) – The route
- **`controller`** (`pycommunicate.server.bases.controller.ControllerFactory`) – The controller factory to associate with the route

`add_error_handler`(*code*, *controller*)

Associates a *`ControllerFactory`* with an HTTP error code

Parameters

- **`code`** (*`int`*) – The error code

- **controller** (pycommunicate.server.bases.controller.ControllerFactory) – The controller factory to associate the code with

New in version 0.0.7.

run()
Run the server, and block the current thread.

2.6 Controller and ControllerFactory

class pycommunicate.server.bases.controller.**ControllerFactory**
This class is used to define how a *Controller* should be created. Its methods are all chainable, or in other words, all return the instance. The constructor has no arguments

add_view(*view*)
Adds a py:class:~*pycommunicate.server.bases.views.View* to the controller factory's view list.

Warning: Adding the same view twice can cause issues

Parameters view (*pycommunicate.server.bases.views.View*) – The view class to add

set_default_view(*view*)
Sets the default view for the controller factory. The default view is the one that is shown initially.

Warning: The client will crash with a 500 server error if this is not set.

Parameters view (*pycommunicate.server.bases.views.View*) – The view class to set as default.

with_before_connect(*before_connect*)
Sets the before_connect function. This is called as soon as a request comes in for the page, and should be used to do something before a page loads.

before_connect takes one argument, an instance of `CallCTX`. This CallCTX contains one function, `abort`, which when called will interrupt the request and send back the error code passed to it.

Parameters before_connect (*function*) – The before_connect function. See above for signature

class pycommunicate.server.bases.controller.**Controller**
The Controller class handles one url, or route. It contains multiple `View` and manages switching between them.

Warning: Do not try and create *Controller* instances on your own. Use *ControllerFactory* for that instead.

route_data
This contains the values of the variable parts in the route. See *add_controller()* for more information on variable route parts.

d
This is the data object, a simple dictionary that use can use to store data across multiple views. It is reset every request, for full sessions across requests use `user.session` instead.

user
An instance of `User` of which this controller is currently servicing. Use its `session` attribute for proper sessions.

`special_return_handler`
If this is not None, then whatever this function returns will be sent directly to flask as the response. Use with caution.

New in version 0.0.7.

`change_view`(*new_view_index*)
Change the active view to the index provided. View indices start at 0 and increase in the order you added them in the controller factory.

Note: This will only do anything if the page has the pycommunicate JS libraries loaded. This function will not work from within a `render()` function.

Parameters **`new_view_index`** (*`int`*) – The new view index to switch to.

`redirect`(*location*)
If called from a child view's `render()` function, this will change the special_return_handler to a function that returns a redirect to the location. Otherwise, it signals the page to redirect elsewhere.

Note: This will only do anything outside of `render()` if the page has the pycommunicate JS libraries loaded.

Parameters **`location`** (*`str`*) – The url to redirect to.

Python Module Index

p

pycommunicate.server.app.communicate, 13

pycommunicate.server.bases.controller, 14

Index

A

add_controller() (pycommunicate.server.app.communicate.CommunicateApp method), 13
add_error_handler() (pycommunicate.server.app.communicate.CommunicateApp method), 13
add_view() (pycommunicate.server.bases.controller.ControllerFactory method), 14

C

change_view() (pycommunicate.server.bases.controller.Controller method), 15
CommunicateApp (class in pycommunicate.server.app.communicate), 13
Controller (class in pycommunicate.server.bases.controller), 14
ControllerFactory (class in pycommunicate.server.bases.controller), 14

D

d (pycommunicate.server.bases.controller.Controller attribute), 14

P

pycommunicate.server.app.communicate (module), 13
pycommunicate.server.bases.controller (module), 14

R

redirect() (pycommunicate.server.bases.controller.Controller method), 15
route_data (pycommunicate.server.bases.controller.Controller attribute), 14
run() (pycommunicate.server.app.communicate.CommunicateApp method), 14

S

set_default_view() (pycommunicate.server.bases.controller.ControllerFactory method), 14
set_secret_key() (pycommunicate.server.app.communicate.CommunicateApp method), 13
special_return_handler (pycommunicate.server.bases.controller.Controller attribute), 14

U

user (pycommunicate.server.bases.controller.Controller attribute), 14

W

with_before_connect() (pycommunicate.server.bases.controller.ControllerFactory method), 14