itty3

Release 1.1.1

Feb 05, 2020

Contents

1 Tutorial **1**
- 1.1 Installation . . . 1
- 1.2 Starting An App . . . 1
- 1.3 Adding Your First View . . . 2
- 1.4 Building A TODO List App . . . 3
- 1.5 Congratulations! . . . 7

2 Extending `itty3` **9**
- 2.1 Custom `HttpResponse` . . . 9
- 2.2 Custom `HttpRequest` . . . 10
- 2.3 Different Routing . . . 10

3 Deploying `itty3` **13**
- 3.1 Deployment Checklist . . . 13
- 3.2 Managing `App.debug` . . . 13
- 3.3 Use A Proper WSGI Server . . . 14

4 Troubleshooting **15**
- 4.1 "My server isn't responding to traffic!" . . . 15
- 4.2 "I'm getting a 500 error and don't know what's wrong!" . . . 15
- 4.3 "The traceback of my error isn't enough!" . . . 15

5 API Documentation **17**
- 5.1 itty3 . . . 17

6 Philosophy **29**
- 6.1 What `itty3` Is . . . 29
- 6.2 What `itty3` Isn't . . . 29
- 6.3 The Future . . . 30

7 Contributing **31**
- 7.1 Ways To Contribute . . . 31
- 7.2 File An Issue/Bug Report . . . 31
- 7.3 Add/Update Documentation . . . 32
- 7.4 Add Test Coverage . . . 32
- 7.5 Add Feature Code . . . 33

Python Module Index **35**

Index **37**

CHAPTER 1

Tutorial

This guide is aimed at getting started with `itty3`. You should be at least basically familiar with Python & HTTP in order to get the most out of this tutorial.

1.1 Installation

For most users in a modern Python environment they can control, you'll want to run:

```
$ pip install itty3
```

Ideally, this should be run within a `virtualenv`/`pipenv`/similar to give you some isolation. Installing Python libraries globally is a recipe for pain & there are many good options out there that are easy/quick to learn to help you avoid dependency hell.

That said, `itty3` is small/minimal by design. The code (minus docstrings & comments) is less than 1k lines of code. This allows you to embed `itty3` directly into your application by copying in `itty3.py` (or even within your app file directly!).

Regardless, `itty3`'s only dependency is Python 3. It's been tested on Python 3.7.4+, but may work on older versions.

Finally, you can find the complete source for this tutorial within the *examples/tutorial_todolist* directory of `itty3`'s source code.

1.2 Starting An App

Once you have installed `itty3`, the next thing you'll want to do is start an application file. A good choice for starting out is to create an empty `app.py` file.

You'll start out by importing `itty3`:

```
import itty3
```

Next, define a module-wide *App* object instance:

```
app = itty3.App()
```

This will allow you start registering "routes", a combination of an HTTP method, a URI path & a view (a callable function that takes a request & produces a response).

After that, provide a way for the built-in server code to run your new `app`. At the bottom of the file, add the following:

```
if __name__ == "__main__":
    app.run()
```

This tells the Python interpreter that, if the file itself is being run by Python, to execute the code within the `if` block. The `app.run()` runs the built-in **development** server & starts listening for requests.

Your complete `app.py` file should now look like:

```
import itty3


app = itty3.App()


if __name__ == "__main__":
    app.run()
```

Now run the server using `python app.py`. You should get a message about the server starting & where it'll handle requests at:

```
itty3 <version>: Now serving requests at http://127.0.0.1:8000...
```

You can now go to the browser & hit http://127.0.0.1:8000/ to access the app!

However, because we haven't written any application-specific code, you're going to get a *404 Not Found* page, which isn't super-useful. Let's fix that...

1.3 Adding Your First View

A "view" function is any callable that can accept a `request` object & produce a `response` object for serving back to the user. We don't need to focus on the details of those for now (more later), so let's just write a basic index view:

```
import itty3


app = itty3.App()


# NEW CODE HERE!
@app.get("/")
def index(request):
    body = "Hello, world!"
    return app.render(request, body)


if __name__ == "__main__":
    app.run()
```

The first line (`@app.get("/")`) is a decorator. That automatically registers your `index` view with the `app`, telling it that the attached view can handle an HTTP `GET` request at the path `/`.

The next line (`def index(request):`) declares the callable that is run when that request is seen. In simple `itty3` applications, that's usually just a plain Python function, but any callable (instance methods, entire classes, etc.) will work. The only guideline is that it should accept a `request` parameter (a `itty3.HttpRequest` object) as the first positional argument.

We then build a simple string body, just to see a customized result out on the page.

The final piece is the `return app.render(request, body)` line. This builds and returns a `HttpResponse` object, which handles all the details of serving a proper HTTP response back to the user, including status codes & headers. `app.render(...)` is simply a convenience function for producing that object.

If you `Ctrl+C` the `python app.py` & restart it, then hit http://127.0.0.1:8000/ in your browser, you should now get a simple page that says `Hello, world!`.

Hurray! Our first real web response works!

Note: You'll note that "render"-ing in the context of `itty3` doesn't do any templating. `itty3` doesn't care how you produce response bodies, as long as a string or iterable can be returned.

This opens a world of options, from reading/returning entire files (good for Single-Page Apps), returning serialized data (like JSON, YAML or XML), or using your own choice of template language (like `Jinja2`, `Mako` or even regular Python format strings).

The downside is that you need to do a bit more work & make a choice around what you want to do. Evaluate your options & choose the one that works for you.

1.4 Building A TODO List App

Let's step beyond this & start crafting a real interactive app. We'll build a very basic TODO list application.

Note: We're going to use a file-based & JSON setup. This is for simplicity in the example code & to avoid further dependencies.

This is suitable for this toy app, but isn't recommended for production unless you know what you're doing. GET YOU AN DATABASE!

We'll leave the beginning code (`import itty3` & `app = itty3.App()`) as well as the ending code (everything after `if __name__ == "__main__":`) alone, focusing only on our application code.

We'll make the index view more useful first.

1.4.1 The Index View

First, let's create a prettier index page. Alongside your `app.py` file, let's create an `index.html` file.

Add the following to that new `index.html` file:

```
<html>
    <head>
        <title>My TODO List</title>
        <style>
```

(continues on next page)

(continued from previous page)

```
            /* Just a quick reset & some basic styles. */
            * { margin: 0; padding: 0; }
            html { background-color: #CCCCCC; text-align: center; }
            body { background-color: #FFFFFF; border-left: 2px solid #999999; border-
↪right: 2px solid #999999; font-family: Helvetica, Arial, sans-serif; font-size:␣
↪14px; margin: 0 auto; text-align: left; width: 60%; padding: 40px 20px; }
            h1, h2, h3 { margin: 10px 0px; font-family: Georgia, 'Times New Roman',␣
↪Times, serif; }
            p { display: block; padding: 10px 0px; }
            ul { display: block; padding: 10px 0px; list-style: none; }
            ul li { border: 1px solid #EEEEEE; padding: 5px; }
            ul li form { display: inline; }
            ul li input { margin: 0px 10px 0px 0px; }
        </style>
    </head>

    <body>
        <header>
            <h1>My TODO List</h1>
        </header>

        <content>
            <ul>
                <!--
                    We'll manually search/replace this out with string formatting.
                    This is where a real template language would come in handy.
                -->
                {{ content }}
            </ul>

            <p>
                <form method="post" action="/create/">
                    <label for="id_todo">Add TODO:</label>
                    <input type="text" id="id_todo" name="todo">
                    <input type="submit" value="Create">
                </form>
            </p>
        </content>
    </body>
</html>
```

Save the file & close it.

Next, alongside the `app.py` & `index.html` files, create a `data.json` file:

```
{
    "items": []
}
```

Save the file & close it.

Now go back to `app.py` & let's update the `index` view to use our new files:

```
# At the top of the file, add:
import json

# ...
```

(continues on next page)

(continued from previous page)

```
# Then update the ``index`` view.
@app.get("/")
def index(request):
    # We'll open/read the HTML file.
    with open("index.html") as index_file:
        template = index_file.read()

    # Pull in the JSON data (currently mostly-empty).
    with open("data.json") as data_file:
        data = json.load(data_file)

    content = ""

    # Create the list of TODO items.
    for offset, item in enumerate(data.get("items", [])):
        # Note: This is gross & dangerous! You need to escape your
        # data in a real app to prevent XSS attacks!
        content += "<li>{}</li>".format(item)

    if not content:
        content = "<li>Nothing to do!</li>"

    # Now "template" in the data.
    template = template.replace("{{ content }}", content)

    # Return the response.
    return app.render(request, template)
```

Restart the server & check http://127.0.0.1:8000/ in your browser. You should now have some HTML & an empty TODO list!

You can verify things are working by manually changing your `data.json` to:

```
{
    "items": [
        "Finish TODO list app",
        "Do some gardening",
        "Take a nap"
    ]
}
```

Then reloading the page (no server restart needed).

1.4.2 Creating New TODOs View

Now that we can see our TODO list, let's add a way to create new TODO. We'll be creating a **second** function:

```
@app.post("/create/")
def create_todo(request):
    # Pull in the JSON data (currently mostly-empty).
    with open("data.json") as data_file:
        data = json.load(data_file)

    # Retrieve the new TODO text from the POST'ed data.
```

(continues on next page)

(continued from previous page)

```
    new_todo = request.POST.get("todo", "---").strip()

    # Append it onto the TODO items.
    data["items"].append(new_todo)

    # Write the data back out.
    with open("data.json", "w") as data_file:
        json.dump(data, data_file, indent=4)

    # Finally, redirect back to the main list.
    return app.redirect(request, "/")
```

We're doing a couple new things here. First, we're using `@app.post(...)`, which hooks up a route for an HTTP POST request.

Second, we're making use of `request.POST`. This is a `QueryDict`, a `dict`-like object that contains all the `POST`'ed form values.

Note: If you're handling JSON or another different request body, you should **NOT** use `request.POST`. Instead, use `request.body` & manually decode the contents of that string.

Finally, we're using `app.redirect(...)`, which is a convenience function for sending an HTTP (temporary) redirect back to the main page. This triggers a fresh load of the TODO list, including the newly added TODO.

Restart your `python app.py`, reload in your browser & try creating a new TODO item.

1.4.3 Aside: Auto-Reloading Server

By now, you may be tired of manually restarting your `app.py` server. To make things a little easier, we'll set up Gunicorn to serve our local traffic.

First, install `gunicorn`:

```
$ pip install gunicorn
```

Next, instead of running `python app.py`, we'll run:

```
$ gunicorn -w 1 -t 0 --reload app:app
```

Now, whenever we change our `app.py` file, `gunicorn` will automatically reload the code. Now it'll always be serving the current code to our browser.

No more restarts!

1.4.4 Marking TODOs As Done

The last bit of our TODO list app is being able to mark a TODO as completed.

First, we'll need to modify our `index` view. Find the line:

```
content += "<li>{}</li>".format(item)
```

. . . and change it to:

```
content += '<li><form method="post" action="/done/{offset}/"><input type="submit"␣
↪value="Complete"></form>{item}</li>'.format(
    offset=offset,
    item=item
)
```

We unfortunately need to use `POST` here, as HTML forms can't submit `DELETE` requests.

Note: Yes, there are better ways to handle this form submission. Adding some modern JS would be a good exercise for the reader. :)

Finally, we need to add a new view:

```
@app.post("/done/<int:offset>/")
def mark_done(request, offset):
    # Pull in the JSON data (currently mostly-empty).
    with open("data.json") as data_file:
        data = json.load(data_file)

    items = data.get("items", [])

    if offset < 0 or offset >= len(items):
        return app.error_404(request, "Not Found")

    # Move it to "done".
    data.setdefault("done", [])
    data["done"].append(items[offset])

    # Slice out the offset.
    plus_one = offset + 1
    data["items"] = items[:offset] + items[plus_one:]

    # Write the data back out.
    with open("data.json", "w") as data_file:
        json.dump(data, data_file, indent=4)

    # Finally, redirect back to the main list.
    return app.redirect(request, "/")
```

This is very similar to the `create_todo` view, with just a couple modifications.

First, note that our path is `/done/<int:offset>/` & we've added a `offset` parameter to the view's function declaration. This lets us capture data out of the URL & use it in our app.

Next, we use that `offset` to check to ensure it's within the bounds of the `items`. If not, we call `app.error_404(...)` to supply a `404 Not Found` response.

We then archive the item to a (potentially new) `"done"` key within our JSON. And then we overwrite the `items` with a slice that excludes the desired offset.

Reload in the browser & give it a spin!

1.5 Congratulations!

With this, you've built your first working application with `itty3`.

For more information, you can:

- Refer to the API docs at *API Documentation*
- Read some example code
- Check out the *Deploying itty3* guide
- Find out how to *Extending itty3*
- Learn how to *Troubleshooting*

Happy developing!

CHAPTER 2

Extending `itty3`

`itty3` is designed to be streamlined & simple to use out of the box. However, it's relatively easy to customize as well.

2.1 Custom `HttpResponse`

In normal usage, `app.render(...)` is simply a shortcut/convenience method for constructing an `itty3.HttpResponse` object. There's nothing extraordinary about these instances, so you can make them yourself if you prefer:

```
import pyyaml

@app.get("/whatev/")
def whatev(request):
    data = {
        "greeting": "Hello",
        "noun": "world",
        "punctuation": "!",
    }
    return itty3.HttpResponse(
        body=pyyaml.dump(data)
        content_type="text/yaml"
    )
```

Or even subclass it to "bake in" complex behavior:

```
class YAMLResponse(itty3.HttpResponse):
    def __init__(self, data, **kwargs):
        body = pyyaml.dump(data)
        kwargs["content_type"] = "text/yaml"
        super().__init__(body, **kwargs)
```

(continues on next page)

(continued from previous page)

```
@app.get("/whatev/")
def whatev(request):
    data = {
        "greeting": "Hello",
        "noun": "world",
        "punctuation": "!",
    }
    return YAMLResponse(data)
```

And you can override/extend `App.render` if you want to automatically use your new subclass:

```
class MyApp(itty3.App):
    def render(self, request, *args, **kwargs):
        # Because of the mismatch in signatures (`body` vs `data`), this
        # may not work perfectly in all situations.
        return YAMLResponse(*args, **kwargs)
```

2.2 Custom `HttpRequest`

As with `HttpResponse`, `itty3.HttpRequest` isn't particularly special. The only interesting/tricky part is automatically constructed by `App.create_request`, so you'll need to override that method when creating a subclass.

For instance, if you wanted to identify a request was secure in a different way (e.g. verifying a nonce was present in headers):

```
class NonceHttpRequest(itty3.HttpRequest):
    def is_secure(self):
        if "X-Secure-Nonce" not in self.headers:
            return False

        nonce = self.headers["X-Secure-Nonce"]
        # Check a DB for that nonce within a time range.

        if not verified:
            return False

        # The request is good. Carry on.
        return True
```

You could then tell your `App` to always use this subclass with:

```
class SuperSecureApp(itty3.App):
    def create_request(self, environ):
        return NonceHttpRequest.from_wsgi(environ)
```

2.3 Different Routing

If `itty3`'s routing doesn't suit you, you can even define your own variant of `Route`. As long as it conforms to a bit of expected API & with a couple minor tweaks to `App`...:

```
# We want to support routes like:
#
#     "/app/:id/:title"
#
# ...and attempt automatic conversion of types.
# This is a naive implementation of that.

class SimpleRoute(object):
    def __init__(self, method, path, func):
        self.method = method
        self.path = path
        self.func = func

    def split_uri(self, path):
        return path.split("/")

    def can_handle(self, method, path):
        if not self.method == method:
            return False

        internal_path_bits = self.split_uri(self.path)
        external_path_bits = self.split_uri(path)

        if len(internal_path_bits) != len(external_path_bits):
            # Without even iterating, we know it's not right.
            return False

        matched = True

        for offset, bit in enumerate(internal_path_bits):
            if not bit.startswith(":"):
                # We're looking for a non-variable, exact match.
                if bit != external_path_bits[offset]:
                    matched = False
                    break
            else:
                # It's a variable. Carry on.
                continue

        return matched

    def extract_kwargs(self, path):
        # This only gets called if the route can handle the URI.
        # So we'll take a shortcut or two here for brevity.
        internal_path_bits = self.split_uri(self.path)
        external_path_bits = self.split_uri(path)

        matches = {}

        for offset, bit in enumerate(internal_path_bits):
            if not bit.startswith(":"):
                # It's not a variable, we don't care.
                continue

            # It's a variable. Slice off the colon.
            var_name = bit[1:]
```

(continues on next page)

(continued from previous page)

```
            # Extract it from the actual URI.
            value = external_path_bits[offset]

            # Try to convert the type.
            try:
                value = int(value)
            except ValueError:
                pass

            try:
                value = float(value)
            except ValueError:
                pass

            if value in ("true", "false"):
                # Store it as a boolean.
                value = value == "true"

            # Finally, track what we found.
            matches[var_name] = value

        return matches


# Now, just override ``App`` to use your new routes.
class SimpleRoutesApp(itty3.App):
    def add_route(self, method, path, func):
        # We swap in the custom class here.
        route = SimpleRoute(method, path, func)
        self._routes.append(route)
```

CHAPTER 3

Deploying `itty3`

3.1 Deployment Checklist

- Ensure `App.debug` is set to `False`
- Use a proper WSGI server
- Check your 404 & 500 pages to ensure no data is leaking
- Set up static asset serving

3.2 Managing `App.debug`

A solid way to manage the `App.debug` option is to rely on environment-based configuration.:

```
import os

import itty3

# Rely on environment variables to set `debug`.
app = itty3.App(
    debug=bool(os.environ.get("APP_DEBUG", False))
)
```

This allows your code to automatically deploy to new environments (such as production) with `debug=False`. For local development, you can export the environment variable to turn debug back on.:

```
$ export APP_DEBUG=1
$ python app.py
```

3.3 Use A Proper WSGI Server

While the built-in `wsgiref` server is convenient, it's not particularly feature-rich or configurable. Some good production-ready alternatives include:

- Gunicorn
- Bjoern
- uWSGI
- CherryPy
- gevent
- Twisted

All are good options, boasting good speed/compliance/configuration across the board. You can't really go wrong, so experiment & find the one you like best.

The author has the most experience with Gunicorn, so example configuration to accommodate that follows:

```
# gunicorn.conf
import multiprocessing

bind = "127.0.0.1:8000"
workers = multiprocessing.cpu_count() * 2 + 1
daemon = True
reload = False
max_requests = 1000
user = "<a-configured-non-root-user>"
# group = "<an-optional-group>"
```

Then running the daemon with:

```
$ gunicorn -c gunicorn.conf myapp:app
```

It's important to note that this runs `gunicorn` still only on the localhost. It's generally advised to put a reverse proxy or similar in front of your WSGI workers.

As an example, here's an nginx setup for proxying to Gunicorn:

```
server {
    listen 80;
    server_name example.org;
    access_log  /var/log/nginx/access.log;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

CHAPTER 4

Troubleshooting

4.1 "My server isn't responding to traffic!"

This is commonly because of the host address `itty3` is bound to. By default, the code & docs default to `127.0.0.1` ("localhost"). This prevents your application server from responding to requests from the outside world.

You'll likely need to change the address to something like `0.0.0.0` ("respond to requests from any host") or have a server that sits in front of your application server (like Nginx or similar).

A way to check if the application server is responding is to be on the server & issue a `curl` command to `http://127.0.0.1:<port-goes-here>/`. If it responds as expected, it's likely an address (or port) issue.

4.2 "I'm getting a 500 error and don't know what's wrong!"

When `App.debug=False` (the default), `itty3` suppresses exceptions & simply returns a static 500 error page.

You'll want to set `App.debug` to `True`, either via:

- the initialization - `itty3.App(debug=True)`
- setting the attribute on the `App` itself - `app.debug = True`
- the `app.run(debug=True)` if you're using that

When you do this, a traceback will be provided in the console output for the server. You can use this to debug your issue.

4.3 "The traceback of my error isn't enough!"

Other frameworks provide more comprehensive or even interactive debug pages. While nice, these come at a complexity & security cost.

Being minimalist, the next best way to figure out what's wrong in your `itty3` application is through the use of an interactive debugger, such as pdb.

You can drop a `import pdb; pdb.set_trace()` pretty much anywhere in your code, step through instructions & examine variable contents.

There are other options, such as `wdb`, `pudb` or even in-editor interactive debuggers! Experiment & find one you like.

Experience with an interactive debugger is an invaluable skillset to have & well worth the time you invest into it.

CHAPTER 5

API Documentation

5.1 itty3

The itty-bitty Python web framework... **Now Rewritten For Python 3!**

class itty3.**App**(*debug=False*)
Bases: object

An orchestration object that handles routing & request processing.

Parameters **debug** (*bool*) – Allows for controlling a debugging mode.

add_route(*method*, *path*, *func*)
Adds a given HTTP method, URI path & view to the routing.

Parameters

- **method** (*str*) – The HTTP method to handle
- **path** (*str*) – The URI path to handle
- **func** (*callable*) – The view function to process a matching request

create_request(*environ*)
Given a WSGI environment, creates a *HttpRequest* object.

Parameters **environ** (*dict-alike*) – The environment data coming from the WSGI server, including request information.

Returns A built request object

Return type *HttpRequest*

delete(*path*)
A convenience decorator for adding a view that processes a *DELETE* request to routing.

Example:

```
app = itty3.App()

@app.delete("/blog/<slug:post_slug>/")
def delete_post(request, post_slug):
    ...
```

Parameters **path** (*str*) – The URI path to handle

error_404(*request*)
Generates a 404 page for when something isn't found.

Exposed to allow for custom 404 pages. **Care** should be taken when overriding this function, as it is used internally by the routing & Python errors bubbling from within can break the server.

Parameters **request** (HttpRequest) – The request being handled

Returns The populated response object

Return type *HttpResponse*

error_500(*request*)
Generates a 500 page for when something is broken.

Exposed to allow for custom 500 pages. **Care** should be taken when overriding this function, as it is used internally by the routing & Python errors bubbling from within can break the server.

Parameters **request** (HttpRequest) – The request being handled

Returns The populated response object

Return type *HttpResponse*

find_route(*method*, *path*)
Determines the routing offset for a given HTTP method & URI path.

Parameters

- **method** (*str*) – The HTTP method to handle
- **path** (*str*) – The URI path to handle

Returns The offset of the matching route

Return type int

Raises ***RouteNotFound*** – If a matching route is not found

get(*path*)
A convenience decorator for adding a view that processes a *GET* request to routing.

Example:

```
app = itty3.App()

@app.get("/blog/<slug:post_slug>/")
def post_detail(request, post_slug):
    ...
```

Parameters **path** (*str*) – The URI path to handle

get_log()
Returns a *logging.Logger* instance.

By default, we return the *itty3* module-level logger. Users are free to override this to meet their needs.

Returns The module-level logger

Return type logging.Logger

patch(*path*)
A convenience decorator for adding a view that processes a *PATCH* (partial update) request to routing.

Example:

```
app = itty3.App()

@app.patch("/blog/bulk/")
def bulk_post(request):
    ...
```

Parameters **path** (*str*) – The URI path to handle

post(*path*)
A convenience decorator for adding a view that processes a *POST* request to routing.

Example:

```
app = itty3.App()

@app.post("/blog/create/")
def create_post(request):
    ...
```

Parameters **path** (*str*) – The URI path to handle

process_request(*environ*, *start_response*)
Processes a specific WSGI request.

This kicks off routing & attempts to find a route matching the requested HTTP method & URI path.

If found, the view associated with the route is called, optionally with the parameters from the URI. The resulting *HttpResponse* then performs the actions to write the response to the server.

If not found, *App.error_404* is called to produce a 404 page.

If an unhandled exception occurs, *App.error_500* is called to produce a 500 page.

Parameters

- **environ** (*dict-alike*) – The environment data coming from the WSGI server, including request information.
- **start_response** (*callable*) – The function/callable to execute when beginning a response.

Returns The body iterable for the WSGI server

Return type iterable

put(*path*)
A convenience decorator for adding a view that processes a *PUT* request to routing.

Example:

```
app = itty3.App()

@app.put("/blog/<slug:post_slug>/")
def update_post(request, post_slug):
    ...
```

Parameters **path** (*str*) – The URI path to handle

redirect(*request*, *url*, *permanent=False*)
A convenience function for supplying a HTTP redirect.

Parameters

- **request** (HttpRequest) – The request being handled
- **url** (*str*) – A path or full URL to redirect the user to
- **permanent** (*bool, Optional*) – Whether the redirect should be considered permanent or not. Defaults to *False* (temporary redirect).

Returns The populated response object

Return type *HttpResponse*

remove_route(*method*, *path*)
Removes a route from the routing.

Parameters

- **method** (*str*) – The HTTP method to handle
- **path** (*str*) – The URI path to handle

render(*request*, *body*, *status_code=200*, *content_type='text/html'*, *headers=None*)
A convenience method for creating a *HttpResponse* object.

Parameters

- **request** (HttpRequest) – The request being handled
- **body** (*str*) – The body of the response
- **status_code** (*int, Optional*) – The HTTP status to return. Defaults to *200*.
- **content_type** (*str, Optional*) – The *Content-Type* header to return with the response. Defaults to *text/html*.
- **headers** (*dict, Optional*) – The HTTP headers to include on the response. Defaults to empty headers.

Returns The populated response object

Return type *HttpResponse*

render_json(*request*, *data*, *status_code=200*, *content_type='application/json'*, *headers=None*)
A convenience method for creating a JSON *HttpResponse* object.

Parameters

- **request** (HttpRequest) – The request being handled
- **data** (*dict/list*) – The Python data structure to be encoded as JSON
- **status_code** (*int, Optional*) – The HTTP status to return. Defaults to *200*.

- **content_type** (*str, Optional*) – The *Content-Type* header to return with the response. Defaults to *text/html*.
- **headers** (*dict, Optional*) – The HTTP headers to include on the response. Defaults to empty headers.

Returns The populated response object

Return type *HttpResponse*

render_static(*request*, *asset_path*)
Handles serving static assets.

WARNING: This should really only be used in development!

It's slow (compared to nginx, Apache or whatever), it hasn't gone through any security assessments (which means potential security holes), it's imperfect w/ regard to mimetypes.

Parameters

- **request** (HttpRequest) – The request being handled
- **asset_path** (*str*) – A path of the asset to be served

Returns The populated response object

Return type *HttpResponse*

reset_logging(*level=20*)
A method for controlling how *App.run* does logging.

Disables *wsgiref*'s default "logging" to *stderr* & replaces it with *itty3*-specific logging.

Parameters **level** (*int, Optional*) – The *logging.LEVEL* you'd like to have output. Default is *logging.INFO*.

Returns

The handler class to be used. Defaults to a custom *NoStdErrHandler* class.

Return type wsgiref.WSGIRequestHandler

run(*addr='127.0.0.1'*, *port=8000*, *debug=None*, *static_url_path=None*, *static_root=None*)
An included development/debugging server for running the *App* itself.

Runs indefinitely. Use *Ctrl+C* or a similar process killing method to exit the server.

Parameters

- **addr** (*str, Optional*) – The address to bind to. Defaults to *127.0.0.1*.
- **port** (*int, Optional*) – The port to bind to. Defaults to *8000*.
- **debug** (*bool, Optional*) – Whether the server should be run in a debugging mode. If provided, this overrides the *App.debug* set during initialization.
- **static_url_path** (*str, Optional*) – The desired URL prefix for static assets. e.g. */static/*. Defaults to *None* (no static serving).
- **static_root** (*str, Optional*) – The filesystem path to the static assets. e.g. *../static_assets*. Can be either a relative or absolute path. Defaults to *None* (no static serving).

class itty3.**HttpRequest**(*uri*, *method*, *headers=None*, *body=''*, *scheme='http'*, *host=''*, *port=80*, *content_length=0*, *request_protocol='HTTP/1.0'*, *cookies=None*)
Bases: object

A request object, representing all the portions of the HTTP request.

Parameters

- **uri** (*str*) – The URI being requested.
- **method** (*str*) – The HTTP method ("GET|POST|PUT|DELETE|PATCH|HEAD")
- **headers** (*dict, Optional*) – The received HTTP headers
- **body** (*str, Optional*) – The body of the HTTP request
- **scheme** (*str, Optional*) – The HTTP scheme ("http|https")
- **host** (*str, Optional*) – The hostname of the request
- **port** (*int, Optional*) – The port of the request
- **content_length** (*int, Optional*) – The length of the body of the request
- **request_protocol** (*str, Optional*) – The protocol of the request
- **cookies** (*http.cookies.SimpleCookie, Optional*) – The cookies sent as part of the request.

GET
Returns a *QueryDict* of the GET parameters.

POST
Returns a *QueryDict* of the POST parameters from the request body.

Useless if the body isn't form-encoded data, like JSON bodies.

PUT
Returns a *QueryDict* of the PUT parameters from the request body.

Useless if the body isn't form-encoded data, like JSON bodies.

content_type()
Returns the received Content-Type header.

Returns The content-type header or "text/html" if it was absent.

Return type str

classmethod from_wsgi(*environ*)
Builds a new HttpRequest from the provided WSGI *environ*.

Parameters **environ** (*dict*) – The bag of YOLO that is the WSGI environment

Returns

A fleshed out request object, based on what was present.

Return type *HttpRequest*

get_status_line()

is_ajax()
Identifies if the request came from an AJAX call.

Returns True if sent via AJAX, False otherwise

Return type bool

is_secure()
Identifies whether or not the request was secure.

Returns True if the environment specified HTTPs, False otherwise

Return type bool

json()
Decodes a JSON body if present.

Returns The data

Return type dict

split_uri(*full_uri*)
Breaks a URI down into components.

Parameters **full_uri** (*str*) – The URI to parse

Returns

A dictionary of the components. Includes *path*, *query* *fragment*, as well as *netloc* if host/port information is present.

Return type dict

class itty3.**HttpResponse**(*body=''*, *status_code=200*, *headers=None*, *content_type='text/plain'*)
Bases: object

A response object, to make responding to requests easier.

A lightly-internal *start_response* attribute must be manually set on the response object when in a WSGI environment in order to send the response.

Parameters

- **body** (*str, Optional*) – The body of the response. Defaults to "".
- **status_code** (*int, Optional*) – The HTTP status code (without the reason). Default is *200*.
- **headers** (*dict, Optional*) – The headers to supply with the response. Default is empty headers.
- **content_type** (*str, Optional*) – The content-type of the response. Default is *text/plain*.

delete_cookie(*key*, *path='/'*, *domain=None*)
Removes a cookie.

Succeed regards if the cookie is already set or not.

Parameters

- **key** (*str*) – The name of the cookie.
- **path** (*str, Optional*) – The path the cookie is valid for. Default is *"/"*.
- **domain** (*str, Optional*) – The domain the cookie is valid for. Default is *None* (only the domain that set it).

set_cookie(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *samesite=None*)
Sets a cookie on the response.

Takes the same parameters as the *http.cookies.Morsel* object from the Python stdlib.

Parameters

- **key** (*str*) – The name of the cookie.
- **value** (*Any*) – The value of the cookie.
- **max_age** (*int, Optional*) – How many seconds the cookie lives for. Default is *None* (expires at the end of the browser session).

- **expires** (*str or datetime, Optional*) – A specific date/time (in UTC) when the cookie should expire. Default is *None* (expires at the end of the browser session).
- **path** (*str, Optional*) – The path the cookie is valid for. Default is *"/"*.
- **domain** (*str, Optional*) – The domain the cookie is valid for. Default is *None* (only the domain that set it).
- **secure** (*bool, Optional*) – If the cookie should only be served by HTTPS. Default is *False*.
- **httponly** (*bool, Optional*) – If *True*, prevents the cookie from being provided to Javascript requests. Default is *False*.
- **samesite** (*str, Optional*) – How the cookie should behave under cross-site requests. Options are *itty3.SAME_SITE_NONE*, *itty3.SAME_SITE_LAX*, and *itty3.SAME_SITE_STRICT*. Default is *None*. Only for Python 3.8+.

set_header(*name*, *value*)

Sets a header on the response.

If the *Content-Type* header is provided, this also updates the value of *HttpResponse.content_type*.

Parameters

- **name** (*str*) – The name of the header.
- **value** (*Any*) – The value of the header.

write()

Begins the transmission of the response.

The lightly-internal *start_response* attribute **MUST** be manually set on the object **BEFORE** calling this method! This callable is called during execution to set the status line & headers of the response.

Returns An iterable of the content

Return type iterable

Raises ***ResponseFailed*** – If no *start_response* was set before calling.

exception itty3.**IttyException**

Bases: Exception

The base exception for all itty3 exceptions.

class itty3.**QueryDict**(*data=None*)

Bases: object

Simulates a dict-like object for query parameters.

Because HTTP allows for query strings to provide the same name for a parameter more than once, this object smoothes over the day-to-day usage of those queries.

You can act like it's a plain *dict* if you only need a single value.

If you need all the values, *QueryDict.getlist* & *QueryDict.setlist* are available to expose the full list.

get(*name*, *default=None*)

Tries to fetch a value for a given name.

If not found, this returns the provided *default*.

Parameters

- **name** (*str*) – The name of the parameter you'd like to fetch

- **default** (bool, defaults to *None*) – The value to return if the *name* isn't found.

Returns The found value for the *name*, or the *default*.

Return type Any

getlist(*name*)
Tries to fetch all values for a given name.

Parameters **name** (*str*) – The name of the parameter you'd like to fetch

Returns The found values for the *name*.

Return type list

Raises **KeyError** – If the *name* isn't found

items()
Returns all the parameter names & values.

Returns

A list of two-tuples. The parameter names & the *first* value for that name.

Return type list

keys()
Returns all the parameter names.

Returns A list of all the parameter names

Return type list

setlist(*name*, *values*)
Sets all values for a given name.

Parameters

- **name** (*str*) – The name of the parameter you'd like to fetch
- **values** (*list*) – The list of all values

Returns None

exception itty3.**ResponseFailed**
Bases: *itty3.IttyException*

Raised when a response could not be returned to the server/user.

class itty3.**Route**(*method*, *path*, *func*)
Bases: object

Handles setting up a given route. Composed of a HTTP method, a URI path & "view" (a callable function that takes a *request* & returns a *response* object) for handling a matching request.

Variables can be added to the path using a <*type:variable_name*> syntax. For instance, if you wanted to capture a UUID & an integer in a URI, you could provide the following *path*:

```
"/app/<uuid:app_id>/version/<int:major_version>/"
```

These would be added onto the call to the view as additional arguments. In the case of the previous *path*, the view's signature should look like:

```
def app_info(request, app_id, major_version): ...
```

Supported types include:

- *str*
- *int*
- *float*
- *slug*
- *uuid*

Parameters

- **method** (*str*) – The HTTP method
- **path** (*str*) – The URI path to match against
- **func** (*callable*) – The view function to handle a matching request

can_handle(*method*, *path*)
Determines if the route can handle a specific request.

Parameters

- **method** (*str*) – The HTTP method coming from the request
- **path** (*str*) – The URI path coming from the request

Returns True if this route can handle the request, False otherwise

Return type bool

convert_types(*matches*)
Takes raw matched from requested URI path & converts the data to their proper type.

Parameters **matches** (*dict*) – The variable names & the *string* data found for them from the URI path.

Returns The converted data

Return type dict

create_re(*path*)
Creates a compiled regular expression of a *path*.

It'd be unusual to need this as an end-user, but who am I to stop you? :)

Parameters **path** (*str*) – A URI path, potentially with <*type:variable_name*> bits in it.

Returns

A tuple of the compiled regular expression that suits the path & dict of the variable names/type conversions to be done upon matching.

Return type tuple

extract_kwargs(*path*)
Pulls variables out of the requested URI path.

Parameters **path** (*str*) – The URI path coming from the request

Returns

A dictionary of the variable names from the path & converted data found for them. Empty dict if no variables were present.

Return type dict

get_re_for_any()

get_re_for_float()

get_re_for_int()

get_re_for_slug()

get_re_for_type(*desired_type*)
Fetches the correct regex for a given type.

Parameters **desired_type** (*str*) – The provided type to get a regex for

Returns A raw string of the regex (minus the variable name)

Return type str

get_re_for_uuid()

known_types = ['str', 'int', 'float', 'uuid', 'slug']

exception itty3.**RouteNotFound**
Bases: *itty3.IttyException*

Raised when no method/path combination could be found.

itty3.**get_version**(*full=False*)
Fetches the current version of itty3.

Parameters **full** (*bool*) – Chooses between the short semver version and the longer/full version, including release information.

Returns The version string

Return type str

CHAPTER 6

Philosophy

The following are the guiding principles around `itty3`:

6.1 What `itty3` Is

- Small (ideally, ~1kloc of code or less)
- Self-contained (depends only on the Python stdlib)
- Open (BSD license, baby!)
- **Respects HTTP & the Web (all the HTTP verbs, good status codes,** content-types, etc.)
- Easy to start working with (import `itty3` & define some functions)
- Flexible & easy to extend (all code can be used in relative isolation)
- Few to no globals (outside of constants)
- Fast & efficient (within reason/readability)
- Well-tested
- Well-documented
- Unicode everywhere (that we can)

`itty3` is designed for the sweet spot around creating single-page apps, small APIs & experiments. The code you produce with `itty3` should be (relatively) easy to migrate to other things (Django, Flask, etc.) when you've moved beyond `itty3`'s capabilities.

6.2 What `itty3` Isn't

- An everything-and-the-kitchen-sink webframework (there are better options)
- **Strongly opinionated about tools** (BYO-database-layer-template-engine-Javascript-framework-etc)

- A perfect solution

`itty3` won't ever ship with an authentication layer, database engines, scaffolding, Makefiles (beyond the Sphinx one), etc.

It's designed for the modern Web, so I'm sure there's ancient things that don't work. Sorrynotsorry.

6.3 The Future

There are planned improvements for the future. The Github Issues for the project is the most up-to-date source of that information, but generally speaking:

- Cookie support
- Included example code
- More/better docs
- Maybe file uploads? *Maybe*

CHAPTER 7

Contributing

`itty3` gladly accepts contributions that:

- are respectful to everyone
- inclusive by default (we're all human)
- report/fix Real World™ problems

If this list offends you or you feel you can't follow these guidelines, you're welcome to use other frameworks or to not contribute.

7.1 Ways To Contribute

In easy-to-harder order:

- File an issue/bug report
- Add/update documentation
- Add test coverage
- Add feature code

7.2 File An Issue/Bug Report

If you encounter a bug or a shortcoming in `itty3`, one of the easiest ways to help is to create a GitHub Issue.

Simply click the link and fill out the template.

The more detail you can provide, the easier it will be to reproduce your issue & get it resolved.

Thanks for the help!

7.3 Add/Update Documentation

Documentation is crucial in understanding & using software libraries. And `itty3`'s documentation is no exception.

That said, mistakes & omissions happen. Helping fix the documentation is an easy way to help everyone!

To submit a documentation addition/update:

- Fork the repository on GitHub
- Clone your fork to your local machine (or edit in the GitHub UI)
- Create a new branch with `git`
- Change the documentation in question (located in the `docs/` folder)
- Add your changes via `git`
- Commit the changes to your branch
- Push the branch back to GitHub
- Open a Pull Request for your branch
- Fill out the template & submit!

With apologies to Edgar Allen Poe,... Quoth the raven: `Documentation forevermore!`

7.4 Add Test Coverage

All software has bugs. Tests prove that the code works as intended.

To submit a test addition/update:

- Fork the repository on GitHub
- Clone your fork to your local machine (or edit in the GitHub UI)
- Create a new branch with `git`
- Create a `virtualenv`/`pipenv` for the repository & activate it
- Run `$ pip install pytest`
- Run `$ pytest tests` & ensure they're passing.
- Make your changes/additions to test files.
- Run `$ pytest tests` again & ensure your changes pass.
- Add your changes via `git`
- Commit the changes to your branch
- Push the branch back to GitHub
- Open a Pull Request for your branch
- Fill out the template & submit!

I `:heart:` more tests, always.

7.5 Add Feature Code

Most everyone loves new features. As long as new features fall within the *Philosophy* of `itty3`, they're welcomed & appreciated!

To submit a new feature:

- Fork the repository on GitHub
- Clone your fork to your local machine (or edit in the GitHub UI)
- Create a new branch with `git`
- Create a `virtualenv`/`pipenv` for the repository & activate it
- Run `$ pip install pytest`
- Run `$ pytest tests` & ensure they're passing.
- Add your feature work.
- Ensure the new feature has a docstring if it's public API.
- Ensure the new feature matches existing code style (`black` is nice here).
- Add documentation to the main guides as appropriate.
- Run `$ pytest tests` again & ensure your changes pass.
- Add your changes via `git`
- Commit the changes to your branch
- Push the branch back to GitHub
- Open a Pull Request for your branch
- Fill out the template & submit!

You da bes!

Python Module Index

i

itty3, 17

Index

A

add_route() (*itty3.App method*), 17
App (*class in itty3*), 17

C

can_handle() (*itty3.Route method*), 26
content_type() (*itty3.HttpRequest method*), 22
convert_types() (*itty3.Route method*), 26
create_re() (*itty3.Route method*), 26
create_request() (*itty3.App method*), 17

D

delete() (*itty3.App method*), 17
delete_cookie() (*itty3.HttpResponse method*), 23

E

error_404() (*itty3.App method*), 18
error_500() (*itty3.App method*), 18
extract_kwargs() (*itty3.Route method*), 26

F

find_route() (*itty3.App method*), 18
from_wsgi() (*itty3.HttpRequest class method*), 22

G

GET (*itty3.HttpRequest attribute*), 22
get() (*itty3.App method*), 18
get() (*itty3.QueryDict method*), 24
get_log() (*itty3.App method*), 18
get_re_for_any() (*itty3.Route method*), 26
get_re_for_float() (*itty3.Route method*), 26
get_re_for_int() (*itty3.Route method*), 27
get_re_for_slug() (*itty3.Route method*), 27
get_re_for_type() (*itty3.Route method*), 27
get_re_for_uuid() (*itty3.Route method*), 27
get_status_line() (*itty3.HttpRequest method*), 22
get_version() (*in module itty3*), 27
getlist() (*itty3.QueryDict method*), 25

H

HttpRequest (*class in itty3*), 21
HttpResponse (*class in itty3*), 23

I

is_ajax() (*itty3.HttpRequest method*), 22
is_secure() (*itty3.HttpRequest method*), 22
items() (*itty3.QueryDict method*), 25
itty3 (*module*), 17
IttyException, 24

J

json() (*itty3.HttpRequest method*), 22

K

keys() (*itty3.QueryDict method*), 25
known_types (*itty3.Route attribute*), 27

P

patch() (*itty3.App method*), 19
POST (*itty3.HttpRequest attribute*), 22
post() (*itty3.App method*), 19
process_request() (*itty3.App method*), 19
PUT (*itty3.HttpRequest attribute*), 22
put() (*itty3.App method*), 19

Q

QueryDict (*class in itty3*), 24

R

redirect() (*itty3.App method*), 20
remove_route() (*itty3.App method*), 20
render() (*itty3.App method*), 20
render_json() (*itty3.App method*), 20
render_static() (*itty3.App method*), 21
reset_logging() (*itty3.App method*), 21
ResponseFailed, 25
Route (*class in itty3*), 25
RouteNotFound, 27

run() (*itty3.App method*), 21

S

set_cookie() (*itty3.HttpResponse method*), 23
set_header() (*itty3.HttpResponse method*), 24
setlist() (*itty3.QueryDict method*), 25
split_uri() (*itty3.HttpRequest method*), 23

W

write() (*itty3.HttpResponse method*), 24