bzz Documentation

Release 0.1.0

Rafael Floriano and Bernardo Heynemann

Nov 15, 2017

Contents

bzz is a rest framework aimed at building restful apis for the tornado web framework.

CHAPTER 1

Getting Started

Installing bzz is as simple as:

```
$ pip install bzz
```

After you have it installed, you need to decide what ORM you'll be using for your models. bzz comes bundled with support for the mongoengine library.

We'll assume you'll be using it for the sake of this tutorial. Let's create our model and our tornado server, then:

```
import tornado.web
from mongoengine import *
import bzz

# just create your own documents
class User(Document):
   __collection__ = "GettingStartedUser"
   name = StringField()

def create_user():
   # let's create a new user by posting it's data
   http_client.fetch(
      'http://localhost:8888/user/',
      method='POST',
      body='name=Bernardo%20Heynemann',
      callback=handle_user_created
   )

def handle_user_created(response):
   # just making sure we got the actual user
   try:
       assert response.code == 200
   finally:
       io_loop.stop()

# bzz includes a helper to return the routes for your models
```

```
# returns a list of routes that match '/user/<user-id>/' and allows for:
# * GET without user-id - Returns list of instances
# * GET with user-id - Returns instance details
# * POST with user-id - Creates new instance
# * PUT with user-id - Updates instance
# * DELETE with user-id - Removes instance
routes = [
    bzz.ModelHive.routes_for('mongoengine', User)
    # and your other routes
]

routes = bzz.flatten(routes)  # making sure tornado gets the correct routes

# Make sure our test is clean
User.objects.delete()

application = tornado.web.Application(routes)
server = HTTPServer(application, io_loop=io_loop)
server.listen(8888)
io_loop.add_timeout(1, create_user)
io_loop.start()
```

CHAPTER 2

Flattening routes

utils.**flatten**(*routes*)

Gets a list of routes that includes hive-generated routes (model, auth or mock), as well as user created routes and flatten the list to the format tornado expects.

Can be used in bzz namespace:

```
import bzz
bzz.flatten(routes)
```

Parameters **routes** – list of routes created by bzz hives or by the user

Returns List of routes that a tornado app expects.

CHAPTER 3

Indices and tables

3.1 Model Hive

bzz kickstarts your development process by handling CRUD operations in your API for your registered models.

Currently bzz supports mongoengine, but support for other ORMs will be done soon. If you would like to see a different ORM supported, please create an issue for it.

3.1.1 What does Model Hive support

- [POST] Create new instances;
- [PUT] Update existing instances;
- [DELETE] Delete existing instances;
- [GET] Retrieve existing instances with the id for the instance;
- [GET] List existing instances (and filter them);

All those operations also work in inner properties. What this means is that if your model has a many-to-one relationship to another model, you get free restful routes to update both.

3.1.2 `ModelHive` class

class `bzz.model.`**`ModelHive`**

classmethod **`routes_for`**(*provider*, *model*, *prefix=''*, *resource_name=None*)

Returns the list of routes for the specified model.

- [POST] Create new instances;
- [PUT] Update existing instances;
- [DELETE] Delete existing instances;

- [GET] Retrieve existing instances with the id for the instance;
- [GET] List existing instances (and filter them);

Parameters

- **provider** (*Full-name provider or built-in provider*) – The ORM provider to be used for this model
- **model** (*Class Type*) – The model to be mapped
- **prefix** (*string*) – Optional argument to include a prefix route (i.e.: '/api');
- **resource_name** (*string*) – an optional argument that can be specified to change the route name. If no resource_name specified the route name is the __class__.__name__ for the specified model with underscores instead of camel case.

Returns route list (can be flattened with bzz.flatten)

If you specify a prefix of '/api/' as well as resource_name of 'people' your route would be similar to:

http://myserver/api/people/ (do a post to this url to create a new person)

Usage:

```
import tornado.web
from mongoengine import *
import bzz

server = None

# just create your own documents
class User(Document):
   __collection__ = "MongoEngineHandlerUser"
   name = StringField()

def create_user():
   # let's create a new user by posting it's data
   http_client.fetch(
      'http://localhost:8890/user/',
      method='POST',
      body='name=Bernardo%20Heynemann',
      callback=handle_user_created
   )

def handle_user_created(response):
   # just making sure we got the actual user
   try:
      assert response.code == 200, response.code
   finally:
      io_loop.stop()

# bzz includes a helper to return the routes for your models
# returns a list of routes that match '/user/<user-id>/' and allows for:
routes = bzz.ModelHive.routes_for('mongoengine', User)

User.objects.delete()
application = tornado.web.Application(routes)
server = HTTPServer(application, io_loop=io_loop)
server.listen(8895)
```

```
io_loop.add_timeout(1, create_user)
io_loop.start()
```

3.1.3 Errors

In the event of a POST, PUT or DELETE, if the model being changed fails validation, a status code of 400 (Bad Request) is returned.

If the model being changed violates an uniqueness constraint, bzz will return a status code of 409 (Conflict), instead.

3.1.4 Supported Providers

`MongoEngine` provider

Provider that supports the rest operations for the MongoEngine ORM.

Allows users to override *get_instance_queryset* and *get_list_queryset* in their models to change how queries should be performed for this provider. Both methods should return a mongoengine queryset and receive as arguments:

- *get_instance_queryset* - model type, original queryset, instance_id and the tornado request handler processing the request
- *get_list_queryset* - original queryset and the tornado request handler processing the request

3.2 Mock Hive

Most of the time creating a new API is a time-consuming task. Other teams (or people) might be depending on your API to create clients or the application that will consume your data.

Usually you'd give them the contract of your API (or worse, they'd have to wait until your API is ready). bzz comes packed with a mocked responses API that allows you to easily craft a mock API that can be used by clients/partners/teams/aliens.

This way you can keep focus in the development of your API, while at the same time allowing people to work with what will eventually be replaced by the real API.

3.2.1 Using Mock Hive

class `bzz.mock.`**`MockHive`**

classmethod **`routes_for`**(*routes_tuple*)
Returns a tuples list of paths, tornado ready

Let's create a new server with a few mocked routes. MockedRoutes expects a list of tuples with

[('METHOD', 'URL or regex', dict(body="string or function", status="200", cookies={'cookie': 'yum'}))]

```
import tornado.web
import bzz

server = None

#first create the routes
```

```
mocked_routes = bzz.MockHive.routes_for([
    ('GET', '/much/api', dict(body='much api')),
    ('POST', '/much/api'),
    ('*', '/much/match', dict(body='such match')),
    ('*', r'/such/.*', dict(body='such match')),
    ('GET', '/much/error', dict(body='WOW', status=404)),
    ('GET', '/much/authentication', dict(body='WOW', cookies={'super': 'cow'})),
    ('GET', '/such/function', dict(body=lambda x: x.method)),
])

def handle_api_response(response):
    # making sure we get the right route
    try:
        assert response.code == 200, response.code
        assert response.body == six.b('much api'), response.body
    finally:
        io_loop.stop()

def get_route():
    # let's test one of them
    http_client.fetch(
        'http://localhost:8891/much/api',
        method='GET',
        callback=handle_api_response
    )

application = tornado.web.Application(mocked_routes)
server = HTTPServer(application, io_loop=io_loop)
server.listen(8891)
io_loop.add_timeout(1, get_route)
io_loop.start()
```

3.3 Auth Hive

bzz comes with decorators and classes to support OAuth2 authentication on specific providers that is easy to plug to your tornado server.

It's worth noting that *AuthHive* is just an extension to the regular tornado routes. That said, it needs to add some dependencies to the application being run.

To enable authentication, call the *AuthHive.configure* method passing your tornado app instance. To get all the routes you need to add to your app, just call *AuthHive.routes_for*:

```
from tornado.web import Application
from tornado.ioloop import IOLoop
import bzz
import bzz.providers.google as google

providers = [
    google.GoogleProvider,
    # MyCustomProvider
]

app = Application(bzz.flatten([
    # ('/', MyHandler),
    bzz.AuthHive.routes_for(providers)
```

```
]))

bzz.AuthHive.configure(app, secret_key='app-secret-key')
```

Note that `flatten()` method encapsulates the list of handlers. It is important because the *routes_for* method returns a list of routes, but *Application* constructor only support routes, so `flatten()` does the magic.

3.3.1 The `AuthHive` class

The bzz framework gives you a AuthHive class to allow easy OAuth2 authentication with a few steps.

class `bzz.auth.`**AuthHive**

The AuthHive is the responsible for integrating authentication into your API.

classmethod **configure**(*app*, *secret_key*, *expiration=1200*, *cookie_name='AUTH_TOKEN'*, *authenticated_create=True*, *authenticated_update=True*, *authenticated_delete=True*, *proxy_host=None*, *proxy_port=None*, *proxy_username=None*, *proxy_password=None*, *authenticated_get=True*)

Configure the application to the authentication ecosystem.

Parameters

- **app** (*tornado.web.Application instance*) – The tornado application to configure
- **secret_key** (*str*) – A string to use for encoding/decoding Jwt that must be private
- **expiration** (*int*) – Time in seconds to the expiration (time to live) of the token
- **cookie_name** (*str*) – The name of the cookie
- **proxy_host** (*str*) – Host of the Proxy
- **proxy_port** (*str*) – Port of the Proxy
- **proxy_username** (*str*) – Username of the Proxy
- **proxy_password** (*str*) – Password of the Proxy
- **authenticated_get** (*bool*) – Should check authentication when listen to *bzz.pre-get-instance* and *bzz.pre-get-list* signals. Default is *True*
- **authenticated_save** (*bool*) – Should check authentication when listen to *bzz.pre-create-instance* signal. Default is *True*
- **authenticated_update** (*bool*) – Should check authentication when listen to *bzz.pre-update-instance* signal. Default is *True*
- **authenticated_delete** (*bool*) – Should check authentication when listen to *bzz.pre-delete-instance* signal. Default is *True*

classmethod **routes_for**(*providers*, *prefix=''*)

Returns the list of routes for the authentication ecosystem with the given providers configured.

The routes returned are for these URLs:

- [prefix]/auth/me/ – To get user data and check if authenticated
- [prefix]/auth/signin/ – To sign in using the specified provider
- [prefix]/auth/signout/ – To sign out using the specified provider

Parameters

- **providers** (*AuthProvider class or instance*) – A list of providers
- **prefix** (*String*) – An optional argument that can be specified as means to include a prefix route (i.e.: '/api');

Returns list of tornado routes (url, handler, initializers)

bzz.auth.**authenticated**(*method*)

Decorate methods with this to require the user to be authenticated.

If the user is not logged in (cookie token expired, invalid or no token), a 401 unauthorized status code will be returned.

If the user is authenticated, the token cookie will be renewed with more *expiration* seconds (configured in *AuthHive.configure* method).

Usage:

```
import tornado
import bzz

class MyHandler(tornado.web.RequestHandler):

    @bzz.authenticated
    def get(self):
        self.write('I`m authenticated! :)')
```

3.3.2 Currently Supported Providers

class bzz.providers.google.**GoogleProvider**(*io_loop=None*)

Provider to perform authentication with Google OAUTH Apis.

authenticate(**args*, ***kwargs*)

Try to get Google user info and returns it if the given access_token get's a valid user info in a string json format. If the response was not an status code 200 or get an error on Json, None was returned.

Example of return on success:

```
{
    id: "1234567890abcdef",
    email: "...@gmail.com",
    name: "Ricardo L. Dani",
    provider: "google"
}
```

3.3.3 Signing-In

In order to sign-in, the authentication route must be called. Both *access_token* and *provider* arguments must be specified. This request must be sent using a *POST* in *JSON* format:

```
POST /auth/signin/ - {'access_token': '1234567890abcdef', 'provider': 'google'}
```

This method returns a 401 HTTP status code if the access_token or provider is invalid.

On success it set a cookie with the name that was specified when you called *AuthHive.configure* (or defaults to AUTH_TOKEN) and returns:

```
200 {authenticated: true}
```

3.3.4 Signing-Out

Signing-out means clearing the authentication cookie. To do that, a *POST* to the sign-out route must be sent:

```
POST /auth/signout/
```

The response clear the authentication cookie and returns:

```
200 {loggedOut: true}
```

3.3.5 Getting User Data

Retrieving information about the authenticated user is as simple as doing a get request to the */me* route:

```
GET /auth/me/
```

If user is not authenticated, the returned value is a *JSON* in this format:

```
200 {authenticated: false}
```

If authenticated:

```
200 {authenticated: true, userData: {}}
```

3.3.6 Authoring a custom provider

Creating a custom provider is as simple as extending `bzz.AuthProvider`. You must override the *authenticate* method.

It receives an *access_token* as argument and should return a dict with whatever should be stored in the JSON Web Token:

```
{
    id: "1234567890abcdef",
    email: "...@gmail.com",
    name: "Ricardo L. Dani",
    provider: "google"
}
```

3.3.7 AuthHive Signals

In order to interact with the authenticated user, you can use the *authorized_user* and *unauthorized_user*:

`signals.`**`authorized_user`** **= None**

This signal is triggered when an user authenticates successfully with the API. The arguments for this signal are *provider_name* and *user_data*. The *provider_name* is used as sender and can be used to filter what signals to listen to. The *user_data* argument is a dict similar to:

```
{
    id: "1234567890abcdef",
    email: "...@gmail.com",
    name: "Ricardo L. Dani",
    provider: "google"
}
```

This is the same dict that's returned in the response to the *authenticate* route. If you'd like to add more fields to the response, just change this dict to include whatever info you'd like to return:

```
@bzz.signals.authorized_user.connect
def handle_authorized(self, provider_name, user_data):
    # let's add something to the user_data dict
    user_data['something'] = 'else'

# now the resulting json for the /authenticate method will be similar to:
{
    "id": "1234567890abcdef",
    "email": "...@gmail.com",
    "name": "Ricardo L. Dani",
    "provider": "google",
    "something": "else"
}
```

signals.**unauthorized_user = None**

This signal is triggered when an user tries to authenticate with the API but fails. The only argument for this signal is *provider_name*. It is used as sender and can be used to filter what signals to listen to.

3.4 Using signals

bzz uses the blinker library for signals. Using them is very simple:

```
import tornado.web
from mongoengine import *
import bzz
from bzz.signals import post_create_instance

server = None

# just create your own documents
class User(Document):
   __collection__ = "GettingStartedUser"
   name = StringField()

def create_user():
   # let's create a new user by posting it's data
   # we ignore the callback and response from http client
   # because we only need the signal in this example.
   http_client.fetch(
      'http://localhost:8889/user/',
      method='POST',
      body='name=Bernardo%20Heynemann'
   )

def handle_post_instance_created(sender, instance, handler):
```

```
    # just making sure we got the actual user
    try:
        assert instance.name == 'Bernardo Heynemann'
    finally:
        io_loop.stop()

# just connect the signal to the event handler
post_create_instance.connect(handle_post_instance_created)

# get routes for our model
routes = bzz.ModelHive.routes_for('mongoengine', User)

# Make sure our test is clean
User.objects.delete()

# create the server and run it
application = tornado.web.Application(routes)
server = HTTPServer(application, io_loop=io_loop)
server.listen(8889)
io_loop.add_timeout(1, create_user)
io_loop.start()
```

3.4.1 Available Signals

pre_get_instance

This signal is sent before an instance is retrieved (GET with a PK).

If a list would be returned the *pre_get_list* signal should be used instead.

Please note that since this signal is sent before getting the instance, the instance is not available yet.

Arguments:

- sender - The model that assigned the signal
- arguments - URL arguments that will be used to get the instance.
- handler - The tornado handler that will be used to get the instance of your model.

Example handler:

```
def handle_pre_get_instance(sender, arguments, handler):
    if handler.application.config.SEND_TO_URL:
        # sends something somewhere
        pass
```

post_get_instance

This signal is sent after an instance is retrieved (GET with a PK).

If a list would be returned the *post_get_list* signal should be used instead.

Arguments:

- sender - The model that assigned the signal
- instance - The instance of your model that was retrieved.

- handler - The tornado handler that was used to get the instance of your model.

Example handler:

```
def handle_post_get_instance(sender, instance, handler):
    # do something with instance
```

pre_get_list

This signal is sent before a list of instances is retrieved (GET without a PK).

If an instance would be returned the *pre_get_instance* signal should be used instead.

Please note that since this signal is sent before getting the list, the list is not available yet.

Arguments:

- sender - The model that assigned the signal
- arguments - URL arguments that will be used to get the instance.
- handler - The tornado handler that will be used to get the instance of your model.

Example handler:

```
def handle_pre_get_list(sender, arguments, handler):
    if handler.application.config.SEND_TO_URL:
        # sends something somewhere
        pass
```

post_get_list

This signal is sent after a list of instances is retrieved (GET without a PK).

If an instane would be returned the *post_get_instance* signal should be used instead.

Arguments:

- sender - The model that assigned the signal
- items - The list of instances of your model that was retrieved.
- handler - The tornado handler that was used to get the instance of your model.

Example handler:

```
def handle_post_get_list(sender, items, handler):
    # do something with the list of items
```

pre_create_instance

This signal is sent before a new instance is created (POST).

Please note that since this signal is sent before creating the instance, the instance is not available yet.

Arguments:

- sender - The model that assigned the signal
- arguments - URL arguments that will be used to create the instance.

- handler - The tornado handler that will be used to create the new instance of your model.

Example handler:

```
def handle_before_instance_created(sender, arguments, handler):
    if handler.application.config.SEND_TO_URL:
        # sends something somewhere
        pass
```

post_create_instance

This signal is sent after a new instance is created (POST).

Arguments:

- sender - The model that assigned the signal
- instance - The instance that was created.
- handler - The tornado handler that created the new instance of your model.

Example handler:

```
def handle_post_instance_created(sender, instance, handler):
    if handler.application.config.SEND_TO_URL:
        # sends something somewhere
        pass

    # do something else with instance
```

pre_update_instance

This signal is sent before an instance is updated (PUT).

Please note that since this signal is sent before updating the instance, the instance is not available yet.

Arguments:

- sender - The model that assigned the signal
- arguments - URL arguments that will be used to update the instance.
- handler - The tornado handler that will be used to update the instance of your model.

Example handler:

```
def handle_before_instance_updated(sender, arguments, handler):
     # if something is wrong, raise error
```

post_update_instance

This signal is sent after an instance is updated (PUT).

Arguments:

- sender - The model that assigned the signal
- instance - The instance that was updated.
- updated_fields - The fields that were updated in the instance with the old and new values.

- handler - The tornado handler that updated the instance of your model.

The *updated_fields* format is like:

```
{
    'field': {
        'from': 1,
        'to': 2
    },
    'field2': {
        'from': 'a',
        'to': 'b'
    }
}
```

Example handler:

```
def handle_post_instance_updated(sender, instance, updated_fields, handler):
    # do something else with instance and/or updated_fields
```

pre_delete_instance

This signal is sent before an instance is deleted (DELETE).

Please note that since this signal is sent before deleting the instance, the instance is not available yet.

Arguments:

- sender - The model that assigned the signal
- arguments - URL arguments that will be used to delete the instance.
- handler - The tornado handler that will be used to delete the instance of your model.

Example handler:

```
def handle_before_instance_deleted(sender, arguments, handler):
    # do something with arguments
```

post_delete_instance

This signal is sent after a new instance is deleted (DELETE).

Arguments:

- sender - The model that assigned the signal
- instance - The instance that was created.
- handler - The tornado handler that created the new instance of your model.

WARNING: The instance returned on this signal has already been removed. How each ORM handles this is peculiar to the given ORM.

Example handler:

```
def handle_post_instance_deleted(sender, instance, handler):
    # do something else with instance
    # just remember the instance has already been deleted!
```

pre_get_user_details

In order to add more items in the authenticated user data (retrieved by */me* route), you can use the *pre_get_user_details*:

signals.**pre_get_user_details = None**

Arguments:

- provider_name: Used as sender and can be used to filter what signals to listen to.
- user_data: A dict similar to:

```
{
    id: "1234567890abcdef",
    email: "...@holmes.com",
    name: "Sherlock Holmes",
    provider: "google"
}
```

- handler: The tornado handler that will be used to update the user data

Example handler:

```
@signals.pre_get_user_details.connect
def handle_pre_get_user_details(self, provider_name, user_data, handler):
    # add user details under user_data, such as:
    # user_data['username'] = 'holmes'
```

- genindex
- modindex
- search

Python Module Index

b

Index