TBone Documentation

Release 0.5.0.4

475 Cumulus Ltd

Jun 03, 2018

Basics

TBone is a framework for building full-duplex, RESTful APIs on top of a Python asynchronous web-server using asyncio.

TBone is web-server agnostic, provided that the web-server is built on *asyncio*. It works out-of-the-box with Sanic or Aiohttp and can be extended for other asyncio-based web-servers as well. TBone was designed to be nonblocking and every component is implemented such that it works with the `asyncio` event loop

CHAPTER 1

Overview

TBone is comprised of 4 major modules:

1. Data Structure - an ODM modeling mechanism for schema definition, data validation and serialization.
2. Data Persistency - Persistency mixin classes for document stores with a full implementation for MongoDB.
3. Resources - Mechanism for creating full-duplex RESTful APIs.
4. Dispatch - Classes for managing internal and external events.

1.1 Motivation

TBone was desighed with a simple goal in mind, to make developing asyncronous web applications and web services easy, quick and painless.

1.1.1 Background

Developers who've gained experience working with frameworks such as Django, often find it difficult to make the switch to develop asyncronous non-blocking web applications and services. This is mainly due to the initial confusion understanding concurrent programming, but also because, at the time of this writing, there is no clear path to quickly and easily develop RESTful APIs that can enjoy the benefits of concurent code and asynchronous web servers.

1.1.2 What TBone tries to solve

TBone was created to make it simple and easy to develop full-duplex RESTful APIs. This means that such APIs have bi-directional communication baked into them, so browser and mobile apps can enjoy efficient communication and a modern user experience. TBone utilizes both HTTP and Websocket protocols to expose REST-like APIs which makes it extremely simple to develop backends for a wide range of applications.

In addition, TBone provides a powerful Object Data Mapper (ODM) layer for schema definition, data validation and serialization, and a persistency layer for MongoDB. Since the ODM layer is decoupled from the persistency layer, it is easy to extend TBone to work with other document stores.

With a REST-like Resource layer, an ODM and a MongoDB persitency layer, TBone makes it possible to develop web applications and services quickly and with a small code footprint.

1.1.3 Minimum Opinions

TBone is an HTTP and Websocket agnostic framework. This means that developers can use either Sanic or Aiohttp as their http/websocket webserver based on their preferences. Its even possible to use polyfill libraries such as SockJS . TBone does not impose an application structure or even a configuration pattern. So you can easily add TBone to your existing async web applications without having to work too hard to fit it in.

1.1.4 Antipatterns

Although TBone strives to be unopinionated as to your application architecture, there are several things which TBone assumes:

1. You are using Python 3.5 and above. At the time of this writing, Python 3.5 is considered a mainstream version of Python. In order to develop quickly and to make testing possible, we have decided not to support Python 2 or Python 3 versions prior to 3.5. This makes the code footprint smaller and easier to maintain.
2. TBone was designed to use asyncio and works with web servers which are built on top of asyncio. Although there are other asyncronous web servers (such as the wonderful Tornado) we chose to stick with asyncio only.

1.2 Installation and Testing

1.2.1 Requirements

The following are requirements to use TBone:

1. TBone requires an asyncronous web server, based on *asyncio* to run on. Out of the box it supports both *Sanic* and *Aiohttp* . It is possible to extend TBone to work with other *asyncio* based web servers
2. TBone works with Python 3.5+ this is due to the use of the async/await syntax. Earlier versions of python will not work.

The easiest way to install TBone is through PyPI:

```
pip install tbone
```

1.2.2 Installation with git

The project is hosted at https://github.com/475cumulus/tbone and can be installed using git:

```
git clone https://github.com/475cumulus/tbone.git
cd tbone
python setup.py install
```

1.2.3 Running the tests

TBone has a suite of tests implemented on top of `pytest` Before running the tests additional requirements need to be installed, including `pytest` and `pytest-asyncio`. The file `test.txt` in the requirements directory lists all requirements needed for testing.

To run all the tests execute the command in the root directory of the project:

```
pytest
```

For coverage results run the following commands:

```
coverage run --source tbone -m py.test
coverage report
```

1.2.4 Optional dependencies

TBone includes very few Python library dependencies. However, depending on the usage developers may need to manually install additional libraries:

Install `sanic` when using TBone with a sanic webserver:

```
pip install sanic
```

Install `aiohttp` when using TBone with a aiohttp webserver:

```
pip install aiohttp
```

To use the `MongoDB` persisrency layer and resources install `Motor` the asynchronous Python driver for MongoDB:

```
pip install motor
```

1.3 Getting Started

Note: Make sure you have at least version 3.5 of Python. TBone uses the async/await syntax, so earlier versions of python will not work.

1.3.1 Selecting a web server

TBone works with either Sanic or Aiohttp . It can be extended to work with other nonblocking web servers. If you're new to both libraries and want to know which to use, please consult their respective documentation to learn which is more suited for your project. Either way, this decision won't affect the you will use TBone.

1.3.2 Model Definition

TBone provides an ODM layer which makes it easy to define data schemas, validate and control their serialization. Unlike ORMs or other MongoDB ODMs, such as MongoEngine, The model definition is decoupled from the data persistency layer, allowing you to use the same ODM with persistency layers on different document stores.

Defining a model looks like this:

```
from tbone.data.fields import *
from tbone.data.models import *

class Person(Model):
    first_name = StringField(required=True)
    last_name = StringField(required=True)
    age = IntegerField()
```

1.3.3 Sanic Example

1.3.4 Aiohttp Example

1.4 Data Structure and Serialization

TBone provides an ODM (Object Document Mapper) for declaring, validating and serializing data structures.

Note: Data structure and data persistency are decoupled in TBone. The ODM is implemented seperately from the persistency layer and thus allows for implementing other datastore persistency layers, in addition to the default one for `MongoDB`

The `Model` class is used as the Base for all data models, with an optional DB mixin class for persistency.

1.4.1 Define a Schema

The ODM works very similarly to `Django` models or other ORMS and ODMs for Python. The main difference is that the classes are not bound, by default, to a datastore. For more information on binding a model to a datastore see *Persistency*

1.4.1.1 Models

Defining a model is done like so:

```
from tbone.data import Model

class Book(Model):
    title = StringField(required=True)
    publication_date = DateField()
    authors = ListField(StringField)
    number_of_pages = IntegerField()
```

Each field in the model is defined by its matching type and by optional parameters which affect its validation and serialization behavior.

Note: Why is TBone not using an external Python schema and validation library such as Marshmallow or Schematics ?

Both libraries mentioned above are excellent for performing the tasks of schema definition, data validation and serialization. However, both libraries were developed to be generic and do not use the asynchronous capabilities of TBone

to their advantage. Therefore TBone implements its own data modeling capabilities which are designed to work in an asynchronous nonblocking environment. An example of this is explained in detail in the *Serialize Methods* section

1.4.1.2 Fields

Fields are used to describe individual variables in a model schema. There are simple fields for data primitives such as `int` and `str` and there are composite fields for describing data such as lists and dictionaries. Developers who have a background in ORM implementations such as the one included in Django, should be very familiar with this concept. All fields classes derive from `BaseField` and implement coersion methods to and from Python natives, with respect to their designated data types. In addition, fields provide additional attributes pertaining to the way data is validated, and the way data is serialized and deserialized. They also provide additional attributes for database mixins

Attributes

The following table lists the different attributes of fields and how they are used

Attribute	Usage	Default
`required`	Determines if the field is required when data is imported or deserialized	`False`
`default`	Declares a default value when none is provided. May be a callable	`None`
`choices`	Set a list of choices, limiting the field's acceptable values	`None`
`validators`	A list of callables to provide external validation methods. See validators	`None`
`projection`	Determines how the field's data is serialized. See Projection	`True`
`readonly`	Determines if data can be deserialized into this field. See Deserialization	`False`
`primary_key`	Used by resources to determine how to construct a resource URI	`False`

There are additional attributes which pertain only to specific fields. For example, `min` and `max` can be defined for an `IntegerField` to determine a range of acceptable values. See the API Reference for more details.

Composite Fields

Composite fields are used to declare lists and dictionaries using the `ListField` and `DictField` fields respectively. A composite field is always based on another field which acts as the base type. A list of integers will be defined as `ListField(IntegerField)` and a dictionary of strings will be defined as `DictField(StringField)` .

The base field which defines the composite field can also accept the standard field attributes. The composite field itself can also define attributes related to its own behavior, like so:

```
class M(Model):
    counters = DictField(IntegerField(default=0))
    tags = ListField(StringField(default='Unknown'), min_size=1, max_size=10)
```

1.4.1.3 Nested Models

Documents can contain nested objects within them. In order to declare a nested object within your model, simply define the nested object as a model class and use the `ModelField` to associate it with your root Model, like so:

```
class Person(Model):
    first_name = StringField()
    last_name = StringField()
```

```
class Book(Model):
    title = StringField(required=True)
    publication_date = DateField()
    author = ModelField(Person)
```

This data model will produce an output like this:

```
{
    "title": "Mody Dick",
    "publication_date" : "1851-10-18",
    "author" : {
        "first_name" : "Herman",
        "last_name" : "Melville"
    }
}
```

Nested objects can also be as the base fields for within lists and dictionaries, like so:

```
class Book(Model):
    title = StringField(required=True)
    publication_date = DateField()
    authors = ListModel(ModelField(Person))
```

This data model will produce an output like this:

```
{
    "title": "The Talisman",
    "publication_date" : "1984-11-08",
    "authors" : [{
        "first_name" : "Stephen",
        "last_name" : "King"
    },{
        "first_name" : "Peter",
        "last_name" : "Straub"
    }]
}
```

Note: If you are using a data persistency mixin such as the `MongoCollectionMixin` you should only add the mixin to your root model and **not** to any of your nested models.

1.4.1.4 Model Options

Every `Model` derived class has an internal `Meta` class which defines its default parameters. This is a very similar approach to meta information declared in Django models.

The following table lists the model options defined within the `Meta` class.

Option	Usage	Default
`name`	Name of the model. This is used in persistency mixins to set the name in the datastores	name of the model
`namespace`	Declares a namespace which prepends the name of the Model	`None`
`exclude_fields`	Exclude fields from base models in subclass	`[]`
`exclude_serialize`	Exclude serialize methos from base model in subclass	`[]`
`creation_args`	Used by `MongoCollectionMixin` for passing creation arguments	`None`
`indices`	Used to declare database indices	`None`

1.4.1.5 Excluding fields and serialize methods

The Model's `Meta` class includes two lists for removing fields and serialize methods inherited from the base class. This is useful when wanting to create multiple resources for the same model, which expose a different set of fields. Consider the following example:

```
class User(Model):
    username = StringField()
    password = StringField()
    first_name = StringField()
    last_name = StringField()

    @serialize
    async def full_name(self):
        return '{} {}'.format(self.first_name, self.last_name)

class PublicUser(User):
    class Meta:
        exclude_fields = ['password']
        exclude_serialize = ['full_name']
```

In this example demonstrates how create a User model, and a PublicUser model, which is a variation of the User model, by inheriting User and then omitting the `password` field and the `full_name` serialize method.

1.4.2 Data Traffic

Models are iterim data components that hold data in memory, coming in and out of the application. Generally, data travels from and to datastores and and application consumers. Models hold the data in memory and facilitate data management in the application flow.

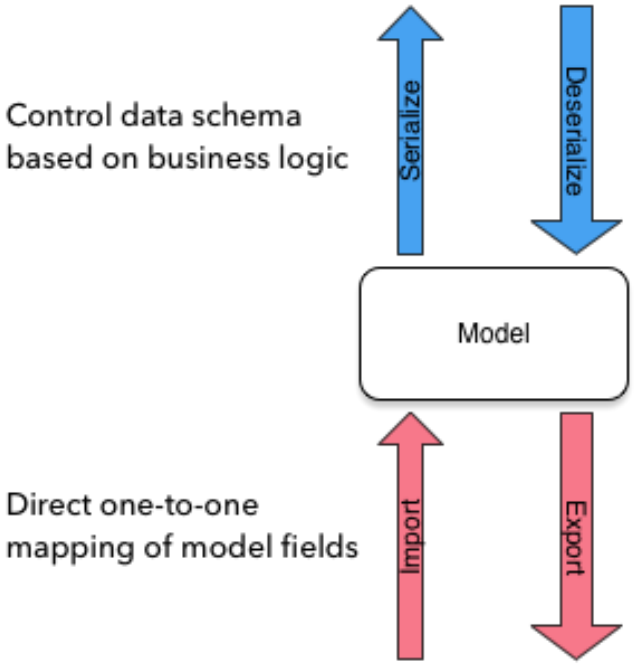
The `Model` class is a central part of TBone and has two data traffic concepts:

1. Import and Export
2. Serialization and deserialization

The big difference between the two data traffic concepts is their purpose. Import and export take data in and out of the `Model` exactly as it is defined in the schema. Serialization and deserialization provides mechanisms for developers to control how data flows in and out of the `Model` to suit the application logic.

Generally speaking, import and export are used for data storage while serialization and deserialization are used for API resources and buiness logic.

The following diagram illustrates this:



It may be useful to consider import / export methods as *inbound* methods, used for storing data in datastores and serialization / deserialization methods as *outbound* methods, used for exposing APIs in a controlled manner

1.4.3 Import Data

There are multiple ways to manipulate data on a `Model`.

The most obvious is to access it's fields directly, like so:

```
>>> book = Book()
>>> b.title = 'Crime and Punishment'
```

While this example is pretty straighforward, it may not be very efficient if in cases were data is already stored in a `dict` which needs to be imported into a `Model`.

The `import_data` method takes care of that, like so:

```
>>> data = {
...     'title': 'Crime and Punishment',
...     'author': ' Fyodor Dostoyevsky',
...     'publication_date': '1866-01-01'  # actual date varies
... }
>>>
>>> book = Book()
>>> book.import_data(data)
```

A quicker way would be to use the `Model` constructor, like so:

```
>> book = Book(data)
```

Data can be imported in a `dict` containing Python types, or data primitives. Once data is imported into the model is coerced into Python types and validated.

1.4.4 Export Data

The `export_data` method is used to convert the model into a Python `dict`. The data is exported in a straighforward manner, mapping all `Model` fields to key/value pairs, like so:

```
>>> data = book.export_data()
>>> data
{'isbn': '9781602523692', 'title': 'War and Peace', 'author': ['Leo Tolstoy'], 'format
↪': 'Paperback', 'publication_date': datetime.datetime(1869, 1, 1, 0, 0,␣
↪tzinfo=tzutc()), 'reviews': [], 'number_of_impressions': 0, 'number_of_views': 0}
>>> type(data)
<class 'dict'>
```

The `export_data` method exports all data in native Python types. It accepts an optional `native` parameter to control how data is exported. If `native` is set to `False` data will be exported in primitive data types, like so:

```
>>> data = book.export_data(native=False)
>>> data
{'isbn': '9781602523692', 'title': 'War and Peace', 'author': ['Leo Tolstoy'], 'format
↪': 'Paperback', 'publication_date': '1869-01-01T00:00:00+00:00', 'reviews': [],
↪'number_of_impressions': 0, 'number_of_views': 0}
>>> type(data)
<class 'dict'>
```

Observing the difference with the previous example where `publication_date` was exported native python `datetime` in this example `publication_date` was exported as a ISO_8601 formatted string.

1.4.5 Validation

Model validation is the process of validating the data contained by the model. Validation is done individually for every field in the Model, and can also include model level validation, to combine values of multiple fields. When `Model.validate` the model iterates through all its fields and call their respective `validate` methods individually. Each type of field implements its own validation, pertaining to its data type.

Explicitly calling the model validation is done like so:

```
m = MyModel({'name': 'ron bugrundy'})
m.validate()
```

The `Model.validate` method does not return any value. However, a `ValueError` exception will be thrown if any validation has failed.

There are 3 forms of field validation:

1. Type validation - Coercing the assigned data to the field's data type.
2. Validator methods - These are field methods which are decorated with `@validator` and perform additional validation that requires logic
3. External validator functions - These are functions which are external to the field class and are passed into field's declaration

To add an external validation to an existing field object, without subclassing, is done like so:

```
def validate_positive(value):
    if value < 0:
        raise ValueError('Value should be positive')

class Person(Model):
    age = IntegerField(validators=[validate_positive])
```

In this example an external validation method was added to the list of validators without subclassing `IntegerField`. This approach is useful when sharing validation methods across different fields.

Another approach is to subclass `IntegerField` and include the validation within the field it self, like so:

```
class PositiveIntegerField(IntegerField):

    @validator
    def positive(value):
        if value < 0:
            raise ValueError('Value should be positive')
```

In this example the validation is implemented within the field's subclass.

1.4.6 Serialization

Models are responsible not only for declaring a schema and validating the data, but also for serializing the models to useful data structures. Controlling the way data models are serialized is extremely useful when creating APIs. More often than not, the application's requirements dictate cases other than a straightforward one-to-one mapping between the data attributes of a model and the API. In some cases there may be a need to omit some data, which is meant only for internal use and not for API consumption. In other cases there may be additional data attributes, required as part of an API endpoint, which are a result of a calculation, aggregation, or data manipulation between 1 or more data attributes.

The following section reviews the tools that are implemented on the `Model` class and how they can be used to yield the desired results.

Model serialization is done using the `serialize` method:

This will produce a Python `dict` with the model's data. Unlike the `export_data` method, the one-to-one mapping of data fields is the default behavior. Developers can use Projection and the `@serialize` decorator to control the serialization of the model

Note: `Model.serialize` is a coroutine, which needs to be awaited, or pushed into the event loop

1.4.6.1 Projection

The previous section went over `Model` serialization methods. This section covers specific instructions that can be added to the `Field` in order to determine how it is serialized.

Every `Field` in the `Model` has a `projection` attribute, which defaults to `True`. The projection field is a ternary value which can be set to either `True`, `False` or `None` and determines the field's serialization in the following way:

1. `True` means that the `Field` will always be serialized, even if its value is `None`
2. `False` means that the `Field` will only be serialized if its value is **not** `None` and will be skipped otherwise.
3. `None` means that the `Field` will never be serialized, regardless of its value.

When a `Model` serialization method is called, it iterates through all the fields and uses the `projection` attribute to determine if and how to serialize the specific field.

The following example illustrates this:

```
>>> from tbone.data.models import *
>>> from tbone.data.fields import *
>>> class BlogPost(Model):
...     title = StringField()
...     body = StringField()
...     number_of_views = IntegerField(default=0, projection=None)
...
>>> post = BlogPost({'title': 'Trees Are Tall', 'body': 'Trees can grow to be very␣
↪tall ...'})
>>> await post.serialize()
{'title': 'Trees Are Tall', 'body': 'Trees can grow to be very tall ...'}
```

Note: Plain Python shell `await` a co-routines as it does not have a running event loop. You can either script this code wrapped as a co-routine or use a 3rd party Python shell which supports an event loop.

The above example illustrates a `Model` that has a field used, in this case, for analytics, and is not required to be included as part of the API

1.4.6.2 `serialize` methods

When designing APIs, it is sometimes required to expose data which is not directly mapped to a single field in the model's schema. Such data can be a result on a calculation, data aggregation or even data fetched from sources outside the model. For this purpose, the `Model` class can implement serialize methods.

Serialize methods are regular member methods on the model with the following attributes:

1. Serialize methods accept no external parameters and rely only on the model's data
2. Serialize methods always return a primitive value
3. Serialize methods are decorated with the `@serialize` decorator
4. Serialize methods are coroutines and therefore are prefixed with `async`

The following example illustrates this:

```
>>> from tbone.data.models import *
>>> from tbone.data.fields import *
>>> class Trainee(Model):
...     weight = FloatField()
```

```
...     height = FloatField()
...     @serialize
...     async def bmi(self): # body mass index
...         return (self.weight*703)/(self.height*self.height)
...
>>> t = Trainee({'weight': 81.5, 'height' : 178})
>>> t.serialize()
{'weight': 81.5, 'height': 178.0, 'bmi': 1.8083101881075623}
```

(Please do not consider the above example to be a real BMI calculator)

The example above brings the quetion of why serialize methods need to be coroutines. In the `bmi` serialize example there are no lines of code which make use of the application's event loop. However, serialize functions may include data from external sources as well. If such an implementation would not be using a coroutine the code will be blocking.

The following example illustrates this:

```
from aiohttp import client
from tbone.data.models import Model
from tbone.data.fields import *

API_KEY = '<get your own for free>';
QUERY_URL = 'http://api.openweathermap.org/data/2.5/forecast?appid={key}&q={city},
↪{state}'

class CityInfo(Model):
    city = StringField()
    state = StringField()

    @serialize
    async def current_weather(self):
        async with aiohttp.ClientSession() as session:
            async with session.get(QUERY_URL.format(key=API_KEY, city=self.city,␣
↪state=self.state)) as resp:
                if resp.status == 200:  # http OK
                    data = await resp.json()
                    return data['list'][0]['main']['temp']
                return None
.
.
.
city_info = CityInfo({'city': 'San Francisco', 'state': 'CA'})
serialized_data = await city_info.serialize()
```

To see a fully working example, please visit the examples page in the project's repository

1.4.7 De-serialization

De-serialization is the process of constructing a data model from raw data, usually passed into the API. The `Model` class implements a `deserialize` method which, by default, matches the data being passed to the fields defined on the model. Variables are assigned to their respective fields and the object's data is validated. Developers may want to customize this behavior to control how models are deserialized, from data.

1.4.7.1 Readonly

Every model field can be assigned with the `readonly` attribute. This tells the model never to accept incoming data to certain fields using the deserialization method. The following example illustrates this:

```
class User(Model):
    username = StringField(required=True)
    password = StringField(readonly=True)
```

1.5 Data Persistency

1.5.1 Overview

TBone provides data persistency in the form of mixin classes. Mixin classes mix with your data models and extend the model's ability to perform CRUD operatons on its data into a datastore. A mixin class is targeted at a specific datastore and implements the underlying functionality over the datastore's API.

Like most parts of TBone, the functionality data persistency mixins should be implemented as nonblocking. Every method which calls upon the database should be implemented as a coroutine. The database driver must support nonblocking calls. Failing to do so will limit's TBone efficiency and your app to be truly asynchorous.

1.5.2 MongoDB

TBone provides the `MongoCollectionMixin` which is a full data persistency mixin implementation over the MongoDB database, using Motor , the asynchronous python driver for MongoDB.

Note: By default TBone does not install the Motor library and its dependency PyMongo. If you're using the `MongoCollectionMixin` for your data persistency you must explicitly install Motor

---

The `MongoCollectionMixin` can be added to your `Model` sub classes like so:

```
from tbone.data.models import *
from tbone.data.fields import *
from tbone.data.fields.mongo import ObjectIdField
from tbone.db.models import MongoCollectionMixin

class Book(Model, MongoCollectionMixin):
    _id = ObjectIdField(primary_key=True)
    isbn = StringField(required=True)
    title = StringField(required=True)
    author = StringField()
    publication_date = DateTimeField()  # MongoDB cannot persist dates only and␣
↪accepts only datetime

    class Meta:
        name = 'books'
        namespace = 'store'  # this will produce a collection named store.books in␣
↪the database
```

In the above example, explicitely defines the `_id` field with the special `ObjectIdField` designed specifically for mongoDB databases. MongoDB will automatically create the `_id` field for every document (unless overruled by

creation arguments) Even if the `_id` field is not explicitely declared in the model. However, developers should add this field to the model to include it in serialization methods.

1.5.2.1 Primary Key

The `primary_key` declared in the example above is **not** used for creating a database index. Its purpose is to set this field as the primary key of the model, for usage in resources. The `MongoResource` class uses the field declared as `primary_key` to construct the resource's URI. A field that is declared as `primary_key` should be unique to the collection. In MongoDB the `_id` field always is and therefore is the default primary key.

There are cases when a different primary key can be defined, that would serve the application's API better. To illustrate this the `Book` example above can be modified slightly, like so:

```
class Book(Model, MongoCollectionMixin):
    _id = ObjectIdField()
    isbn = StringField(primary_key=True)
    title = StringField(required=True)
    author = StringField()
    publication_date = DateTimeField()  # MongoDB cannot persist dates only and␣
↪accepts only datetime

    class Meta:
        name = 'books'
        namespace = 'store'  # this will produce a collection named store.books in␣
↪the database
        indices = [{
            'name': '_isbn',
            'fields': [('isbn', pymongo.ASCENDING)],
            'unique': True
        }]
```

Fields that are declared as the primary key must have an index created with a unique constraint. For more declaring indices see *Indices* below.

The `MongoResource` class automatically identifies the field designated at the primary and adjust its resource URI construction accordingly. The API would then be accessed like so:

```
/api/books/9788422503552/
```

Passing the book's ISBN as the resource's unique identifier.

1.5.2.2 Indices

Each `Model` subclass can define a list of index directives which can be applied to the database's collection. By default MongoDB creates a default index to the `_id` field which is assigned to every document. MongoDB provides an extensive list of features related to document indices. To learn more about MongoDB's indices see the MongoDB documentation.

TBone provides a convinient way to declare indices in the Model's `Meta` class, which adhere to the MongoDB index rules.

The following shows an example:

```
class Book(Model, MongoCollectionMixin):
    isbn = StringField(primary_key=True)
    title = StringField(required=True)
    author = ListField(StringField)
```

```
    class Meta:
        name = 'books'
        indices = [{
            'name': '_isbn',
            'fields': [('isbn', pymongo.ASCENDING)],
            'unique': True
        }]
```

This example of a `Model` subclass mixed with the `MongoCollectionMixin` The `Meta` class includes one index directive with the following attributes: 1. name : give the index a unique name 2. fields: a list of fields to use for creating the index 3. unique: indicate that the field's value (isbn in this case) must be unique

It is important to remember that, unlike ORMs for relational databases, TBone model indices are **not** created automatically. There is no concept of data migration and table (or collection) creation. In fact, MongoDB automatically creates a new collection when writing a document into a non-existing collection. Therefore, it is up to the developer to **explicitly** call TBone's model creation method for every model in the app. This is done with the `create_collection` function

Calling the `create_collection` function for every model is something that should be done only when changes are made to the model's indices or when deploying to a new system. Therefore, a common practice would be to include an additional Python script to achieve this. Please note that `create_collection` is a coroutine and needs to be executed within an event loop:

```
#!/usr/bin/env python
# encoding: utf-8

import asyncio
from bson.json_util import loads
from tbone.db import connect
from tbone.db.models import create_collection
from app import db_config
from models import Book, Author, Publisher

async def bootstrap_db():
    db = connect(**db_config)

    futures = []
    for model_class in [Book, Author, Publisher]:
        futures.append(create_collection(db, model_class))

    await asyncio.gather(*futures)

def main():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(bootstrap_db())
    loop.close()

if __name__ == "__main__":
    main()
```

1.5.2.3 Additional Database Operations

The `MongoCollectionMixin` mainly provides methods for performing CRUD database operations. However, the MongoDB API provides a vast number of tools and methodologies to implements all kinds of data manipulation scenarios. The following example demonstates such a case:

```
class Review(Model):
    user = StringField(required=True)
    text = StringField(required=True)


class Book(Model, MongoCollectionMixin):
    isbn = StringField(primary_key=True)
    title = StringField(required=True)
    author = ListField(StringField)
    publication_date = DateTimeField()  # MongoDB cannot persist dates only and␣
↪accepts only datetime
    reviews = ListField(ModelField(Review), default=[])
```

In this example there is a `Book` model which contains a field that is a list of reviews. This list is essentially a list of embedded documents, defined in the `Review` model. This is one of the ways to implement a one-to-many relationship with a document store, such as MongoDB, by embedding all the reviews inside the book document itself. If this was implemented with a relational database, most likely the `Review` model was an independent table and each record in this table would have a foreign-key to a record in the `Book` table. Therefore, adding a new review would be a single database operation to insert a new record to the Review table.

But in a document store, with reviews embedded into the book document, using basic CRUD database operations the following needs to be done: 1. Fetch the book document 2. Append a new review to the list of embedded review documents (allowing unrestrained access to the whole list) 3. Saving the book document back to the database

This seems to be a lot of work for a simple insertion of one review, not to mention the exposure to data that was otherwise inserted by other users. To solve this, MongoDB provides the `$push` operator, which enables the appending of a single embedded document into the review list. This can be done in a single database operation without having to fetch the whole document first.

In order to utilize this capability the `Book` Model is extended with an additional custom method for performing this operation, like so:

```
class Book(Model, MongoCollectionMixin):
    isbn = StringField(primary_key=True)
    ...


    async def add_review(self, db, review_data):
        ''' Adds a review to the list of reviews, without fetching and updating the␣
↪entire document '''
        db = db or self.db
        # create review model instance
        new_rev = Review(review_data)
        data = new_rev.export_data(native=True)
        # use model's pk as query
        query = {self.primary_key: self.pk}
        # push review
        result = await db[self.get_collection_name()].update_one(
            filter=query,
            update={'$push': {'reviews': data}},
        )
        return result
```

This model's custom-made method takes care of adding a new review to the document with a single database operation and without exposing the entire model to a full-document update.

MongoDB provides many operators that can be used to extend the basic CRUD methodology and thus improve code reliability and performance. Please consult the MongoDB documentation to learn more about operators.

1.5.2.4 Full Text Search

TBone provides out-of-the-box full text search capabilities over MongoDB collections, accessed through the API. Resource which subclass `MongoResource` already have most of the wiring to execute full text search on their data. In order to utilize the full text search capabilities, the Model needs to include an index for FTS like so:

```
class Movie(Model, MongoCollectionMixin):
    _id = ObjectIdField(primary_key=True)
    title = StringField(required=True)
    plot = StringField()
    director = StringField()
    cast = ListField(StringField)
    release_date = DateField()
    runtime = IntegerField()
    poster = URLField()
    genres = StringField()

    class Meta:
        indices = [
            {
                'name': '_fts',
                'fields': [
                    ('title', pymongo.TEXT),
                    ('plot', pymongo.TEXT),
                    ('cast', pymongo.TEXT),
                    ('genres', pymongo.TEXT)
                ]
            }
        ]

class MovieResource(SanicResource, MongoResource):
    class Meta:
        object_class = Movie
```

Once the FTS index is created and indexing is complete, searching the database through the api can simply be done by making an HTTP request, like so:

```
/api/movies/?q="Robert De Niro"
```

This request will yield all the results that include this search phrase in either of the fields that were indexed for FTS. For more on MongoDB full text search, see the MongoDB documentation

The `q` operand is used by default, but can be replaced. Doing so requires a change to the `fts_operator` in the resources's `Meta` class, like so:

```
class MovieResource(SanicResource, MongoResource):
    class Meta:
        object_class = Movie
        fts_operator = 'search'
```

Then, the making the HTTP request is done like so:

```
/api/movies/?search="Robert De Niro"
```

1.5.3 Extending to other datastores

MongoDB is a general-purpose NoSQL document store that has been around for a while. It is widely used as an alternative to relational databases and offers a wide range of features. Due to various considerations, developers may choose to use a different database that is more tuned to their application requirements. TBone provides a MongoDB persistency layer for models, but that layer can be replaced with a custom solution for another database. Not all NoSQL databases would generally merge easily with TBone's ODM. However, most NoSQL document-oriented and key-value databases should be easily integrated with the ODM paradigm.

1.6 Resources

Resources are at the heart of the TBone framework. They provide the foundation for the application's communication with its consumers and facilitate its API. Resources are designed to implement a RESTful abstraction layer over HTTP and Websockets protocols and assist in the creation of your application's design and infrastructure.

1.6.1 Overview

Resources are class-based. A single resource class implements all the methods required to communicate with your API over HTTP or Websocket, using HTTP-like verbs such as `GET` and `POST`. In addition it implements resource events which translate to application events sent over websockets to the consumer.

A `Resource` subclass must implement all the methods it expects to respond to. The following table lists the HTTP verbs and their respective member methods:

HTTP Verb	`Resource` subclass method to implement
GET	`list()`
GET <pk>	`detail()`
POST	`create()`
POST <pk>	`create_detail()`
PUT	`update_list()`
PUT <pk>	`update()`
PATCH	`modify_list()`
PATCH <pk>	`modify()`
DELETE	`delete_list()`
DELETE <pk>	`delete()`

1.6.2 Sanic and AioHttp

TBone includes two mixin classes to adapt your resources to the underlying web-server of your application. Those are `SanicResource` and `AioHttpResource` . Every resource class in your application must include one of those mixin classes, respective to your application's HTTP and Websockets infrastructure. These mixin classes implement the specifics pertaining to their respective libraries and leave the developer with the work on implementing the application's domain functionality.

If your application is based on `Sanic` your resources will be defined like so:

```
class MyResource(SanicResource, Resource):
    ...
```

If your application is based on `AioHttp` your resources will be defined like so:

```
class MyResource(AioHttpResource, Resource):
    ...
```

Note: Adapting a resource class is done with mixins rather than with single inheritance. The reason is so developers can bind the correct resource adapter to a `Resource` derived class or classes that are derived from other base resources such as `MongoResource` . It obviously makes no sense to have resources mixed with both `SanicResource` and `AioHttpResource` in the same project.

1.6.3 Resource Options

Every resource has a `ResourceOptions` class associated with it, that provides the default options related to the resource. Such options can be overriden using the `Meta` class within the resource class itself, like so:

```
from tbone.resources import Resource

class MyResource(Resource):
    class Meta:
        allowed_detail = ['get', 'post']  # In this example, only GET and POST␣
↪methods are allowed
```

Resource options are essential to resources who wish to override built-in functionality such as:

- Serialization
- Authentication
- Allowed methods

For a full list of resource options see the *API Reference*

1.6.4 Formatters

Formatters are classes which help to convert Python `dict` objects to text (or binary), and back, using a certain transport protocol. In TBone terminology, formatting turns an native Python object into another representation, such as JSON or XML. Parsing is turning JSON or XML into native Python object.

Formatters are used by resource objects to convert data into a format which can be wired over the net. When using the HTTP protocol, generally APIs expose data in a text-based format. By default, TBone formats and parses objects to and from a JSON representation. However, developers can override this behavior by writing additional `Formatter` classes to suit their needs.

1.6.5 Authentication

TBone provides an authentication mechanism which is wired into the resource's flow. All requests made on a resource are routed through a central `dispatch` method. Before the request is executed an authentication mechanism is activated to determine if the request is allowed to be processed. Therefore, every resource has an `Authentication` object associated with it. This is done using the `Meta` class of the resource, like so:

```
class BookResource(Resource):
    class Meta:
        authentication = Authentication()
```

By default, all resources are associated with a `NoAuthentication` class, which does not check for any authentication whatsoever. Developers need to subclass `NoAuthentication` to add their own authentication mechanism. Authentication classes implement a single method `is_authenticated` which has the request object passed. Normally, developers would use the request headers to check for authentication and return `True` or `False` based on the content of the request.

1.6.6 HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) is part of the REST specification. TBone supports basic HATEOAS directives and allows for extending this support in resource subclasses. By default, all TBone resources include a `_links` key in their serialized form, which contains a unique `href` to the resource itself, like so:

```
{
    "first_name': 'Ron",
    "last_name': 'Burgundy",
    "_links" : {
        "self" : {
            "href" : "/api/person/1/"
        }
    }
}
```

Disabling HATEOAS support is done per resource, by setting the `hypermedia` flag in the `ResourceOptions` class to `False`, like so:

```
class NoHypermediaPersonResource(Resource):
    class Meta:
        hypermedia = False
    ...
```

Adding additional links to the resource is done by overriding `add_hypermedia` on the resource subclass.

1.6.7 Nested Resources

Nested resources is a technique to extend a resource's endpoints beyond basic CRUD. Every resource automatically exposes the HTTP verbs (GET, POST, PUT, PATCH, DELETE) with their respective methods, adhereing to REST principles. However, it is sometimes neccesary to extend a resource's functionality by implementing additional endpoints. These can be described by two categories:

1. Resources which expose nested resources classes
2. Resources which expose additional unrest endpoints serving specific functionality.

Lets look at some examples:

```
# model representing a user's blog comment. Internal
class Comment(Model):
    user = StringField()
    content = StringField()

# model representing a single blog post, includes a list of comments
class Blog(Model):
    title = StringField()
    content = StringField()
    comments = ListField(ModelField(Comment))
```

```
class CommentResource(ModelResource):
    class Meta:
        object_class = Comment


class BlogResource(ModelResource):
    class Meta:
        object_class = Blog

    @classmethod
    def nested_routes(cls, base_url):
        return [
            Route(
                path=base_url + '%s/comments/add/' % (cls.route_param('pk')),
                handler=cls.add_comment,
                methods=cls.route_methods(),
                name='blog_add_comment')
        ]

    @classmethod
    async def add_comment(cls, request, **kwargs):
```

1.6.8 MongoDB Resources

The `MongoResource` class provides out-of-the-box CRUD functionality over your MongoDB collections with as little as three lines of code, like so:

```
from tbone.resources.mongo import MongoResource

class BookResource(AioHttpResource, MongoResource):
    class Meta:
        object_class = Book
```

Important: TBone is not aware of how you manage your application's global infrastructure. Therefore Resources and Models are not aware of your database's handle. Because of that, TBone makes the assumption that your global `app` object is attached to every `request` object, which both `Sanic` and `AioHttp` do by default. it also assumes that the database handler is assigned to the global `app` object, which you must handle yourself, like so:

```
app.db = connect(...)
```

See TBone examples for more details

1.6.8.1 CRUD

The `MongoResource` class provides out-of-the-box CRUD operations on your data models. As mentioned in the *Persistency* section, models are mapped to MongoDB collections. This allows for HTTP verbs are to be mapped directly to a MongoDB collection's core functionality.

The following table lists the way HTTP verbs are mapped to MongoDB collections

HTTP Verb	MongoDB Collection method
GET	`find()` `find_one()`
POST	`insert()`
PUT	`save()`
PATCH	`find_and_modify()`
DELETE	`delete()`

1.6.8.2 Filtering

The `MongoResource` provides a mapping mechanism between url parameters and MongoDB query parameters. Therefore, the url:

```
/api/v1/movies/?genre=drama
```

Will be mapped to:

```
coll.find(query={"genre": "drama"})
```

Passing additional parameters to the url will add additional parameters to the query.

In addition, it is possible to also add the query operator to the urls parameters. Operators are added to the url parameters using a double underscore __ like so:

```
/api/v1/movies/?rating__gt=4
```

Which will be mapped to:

```
coll.find(query={{"rating": {"$gt": 4}})
```

1.6.8.3 Sorting

Sorting works very similar to filtering, by passing url parameters which are mapped to the sort parameter like so:

```
/api/v1/member/?order_by=age
```

Which will be mapped to:

```
coll.find(sort={'age': 1})  # pymongo.ASCENDING
```

Placing the - sign befor ethe sorted field's name will sort the collection in decending order like so:

```
/api/v1/member/?order_by=-age
```

Which will be mapped to:

```
coll.find(sort={'age': -1})  # pymongo.DESCENDING
```

1.6.8.4 Full Text Search

The `MongoResource` class provides an easy hook between url parameters and a full-text-search query. However, full text search is not available on a collection by default. In order to utilize MongoDB's FTS functionality the proper indices must be configured within the collection. Please consult with the MongoDB documentation on using text indices as well as TBone's documentation on defining indices as part of a `Model` .

FTS (full text search) is provided out-of-the-box on all `MongoResource` classes, provided the relevant indices are in place. FTS can be used using query parameters like so:

```
/api/books/?q=history
```

This will execute a FTS query on all fields that were indexed with the text index. FTS takes presedence over standard filters, which means that if the url parameters include both FTS and filters, FTS will be executed.

The default operator for accessing FTS is `q`. However, this can overriden in the `Meta` class by overriding the option `fts_operator` like so:

```
class BookResource(SanicResource, MongoResource):
    class Meta:
        object_class = Book
        fts_operator = 'fts'
```

This will result in a usage like so:

```
/api/books/?fts=history
```

1.6.9 Hooking up to application's router

Once a resource has been implemented, it needs to be hooked up to the application's router. With any web application such as Sanic or AioHttp, adding handlers to the application involves matching a uri to a specific handler method. The `Resource` class implements two methods `to_list` and `to_detail` which create list handlers and detail handlers respectively, for the application router, like so:

```
app.add_route('GET', '/books', BookResource.as_list())
app.add_route('GET', '/books/{id}', BookResource.as_detail())
```

The syntax varies a little, depending on the web server used.

1.6.9.1 Sanic Example

```
from sanic import Sanic
from tbone.resources import Resource
from tbone.resources.sanic import SanicResource


class TestResource(SanicResource, Resource):
    async def list(self, **kwargs):
        return {
            'meta': {},
            'objects': [
                {'text': 'hello world'}
            ]
        }

app = Sanic()
app.add_route(methods=['GET'], uri='/', handler=TestResource.as_list())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

1.6.9.2 AioHttp Example

```
from aiohttp import web
from tbone.resources import Resource
from tbone.resources.aiohttp import AioHttpResource


class TestResource(AioHttpResource, Resource):
    async def list(self, **kwargs):
        return {
            'meta': {},
            'objects': [
                {'text': 'hello world'}
            ]
        }

app = web.Application()
app.router.add_get('/', TestResource.as_list())

if __name__ == "__main__":
    web.run_app(app, host='127.0.0.1', port=8000)
```

The examples above demonstrate how to manually add resources to the application router. This can become tedious when the app has multiple resources which expose list and detail endpoints as well as some nested resources. An alternative way is to use a `Router` , described below.

1.6.10 Routers

Routers are *optional* components which help to bind resources to the application's url router. Whether you're using `Sanic` or `AioHttp` every application must have its url routes defined.

The fact that AioHttp uses a centralized system of defining routes, similar to `Django`, while Sanic uses a decentralized system of defining routes, in the form of decorators, bears no difference.

Resources are registered with routers. A router may have one or more resources registered with it. An application can have one or more routers defined.

Note: For small applications a single router for all your resources may be good enough. Larger applications may want to use multiple routers in order to seperate the application's components, similar to the way a Django project may contain multiple apps. It is up to the developers to decide how many routes are needed in their projects.

A router may have an optional `path` variable which the router prepends to all resources.

Resources are registered with a router like so:

```
class AccountResource(AioHttpResource, Resource):
    ...

class PublicUserResource(AioHttpResource, Resource):
    ...

router = Router(name='api/user')                    # api/user is the url prefix of␣
↪all resources under this router
router.register(AccountResource, 'account')         # the full url would be api/user/
↪account/
router.register(PublicUserResource, 'public_user')  # the full url would be api/user/
↪public_user/
```

Once the router is created, the urls need to be added to the application's urls.

With `AioHttp` it looks like this:

```
app = web.Application()
.
.
.
for route in router.urls():
    app.router.add_route(
        method=route.methods,
        path=route.path,
        handler=route.handler,
        name=route.name
    )
```

With `Sanic` it looks like this:

```
app = Sanic()
.
.
.
for route in router.urls():
    app.add_route(
        methods=route.methods,
        uri=route.path,
        handler=route.handler
    )
```

1.7 Dispatch

The dispatch sub-mobule consists of all the classes used for managing internal and external events.

1.7.1 Signals

Signals are a classes which help implement a publish/subscribe, or producer/consumer relationship between components in the application. The signals mechanism is very similar to the same concept existing in frameworks such as Django. However, signals in TBone are implemented to be non-blocking. Because of that, signals not only help to build code with the separation of concerns principle, but also allow application internal events to execute outside the scope of the request/response cycle, a pattern often implemented with background tasks.

1.7.1.1 Declaring

signals are declared like so:

```
from tbone.dispatch import Signal

post_init = Signal()
```

Any module within the app can import this `Signal` object and can register as a receiver or use it to send events to other receiver methods

Note: The `MongoCollectionMixin` uses the `post_save` and `post_delete` events to signal that a documentent as been inserted, updated or deleted from the database. Components using `MongoCollectionMixin` based models, such as the `MongoResource` can consume such events to implement further functionality

Signals can be triggered with parameters such as `sender` and `instance` and any other parameter that is required to pass to the receiving method. Since signals are only handled within the same process, it is safe to pass Python objects.

1.7.1.2 Sending Signals

Sending a signal is done like so:

```
import asyncio
from my_signals import post_init  # a module containing the declaration of the signal

...

asyncio.ensure_future(post_init.send(sender=App))
```

By using the `ensure_future` method of `asyncio` the `send` method is injected into the event loop without awaiting on it. If the call to `send` was part of a request/response cycle, the execution of the receiver methods will most likely happen after the response is returned to the client. Using `ensure_future` is not a requirement. Signals can be executed synchronously. The usage entirely depends on the developers' intentions.

1.7.1.3 Receiving Signals

Receiving, or consuming signals requires implementing a method and then registering this method as the signal receiver function using the `Signal.connect()`

`Signal.`**`connect`**(*receiver*, *sender*)

Connects a signal to a receiver function

Parameters

- **`receiver`** – The callback function which will be connected to this signal
- **`sender`** – Specifies a particular sender to receive signals from. Used to limit the receiver function to signal from particular sender types

Calling the `connect` method is done like so:

```
from my_signals import post_init

def on_init(sender, **kwargs):
    ... do something

post_init.connect(on_init)
```

1.7.2 Channels

Channels are another form of implementing a publish/subscribe relationship between software components, but is intended for external communication. The most common use of channels in TBone is for server-to-client communication via websockets. Channels provide the neccesary abstraction between code in the app and open websockets. Tbone provides the flexibility of creating multiple channels directing messages on the same websocket, for publishing different elements of the application.

Channels can store message data in any medium, depending on the implementation. TBone includes two implementations out of the box:

1. `MemoryChannel` : Uses an `asyncio.Queue` for managing message data. Useful *only* when deploying a single instance of the backend
2. `MongoChannel` : Uses a `MongoDB` capped collection and a tailable cursor to wait for new messages. This technique can be very useful for backends consisting of multiple instances. It is also quite useful for TBone-based apps which use `MongoDB` as the data store, since no additional component is required. However, for large volumes the performance cannot match that of a RAM-based database such as `memcached` or `Redis`

Custom-backend channels can be implemented by subclassing the `Channel` class.

Unlike signals, a single channel respond to multiple types of events. The `subscribe` method is used to register to channel events.

`Channel.`**`subscribe`**(*event*, *subscriber*)
Subscribe to channel events.

Parameters

- **`event`** – The name of the event to subscribe to. String based
- **`subscriber`** – A `Carrier` type object which delivers the message to its target

Sending events to channel subscribers is done with the `publish` method:

`Channel.`**`publish`**(*key*, *data=None*)
Publish an event to the channel, to be sent to all subscribers

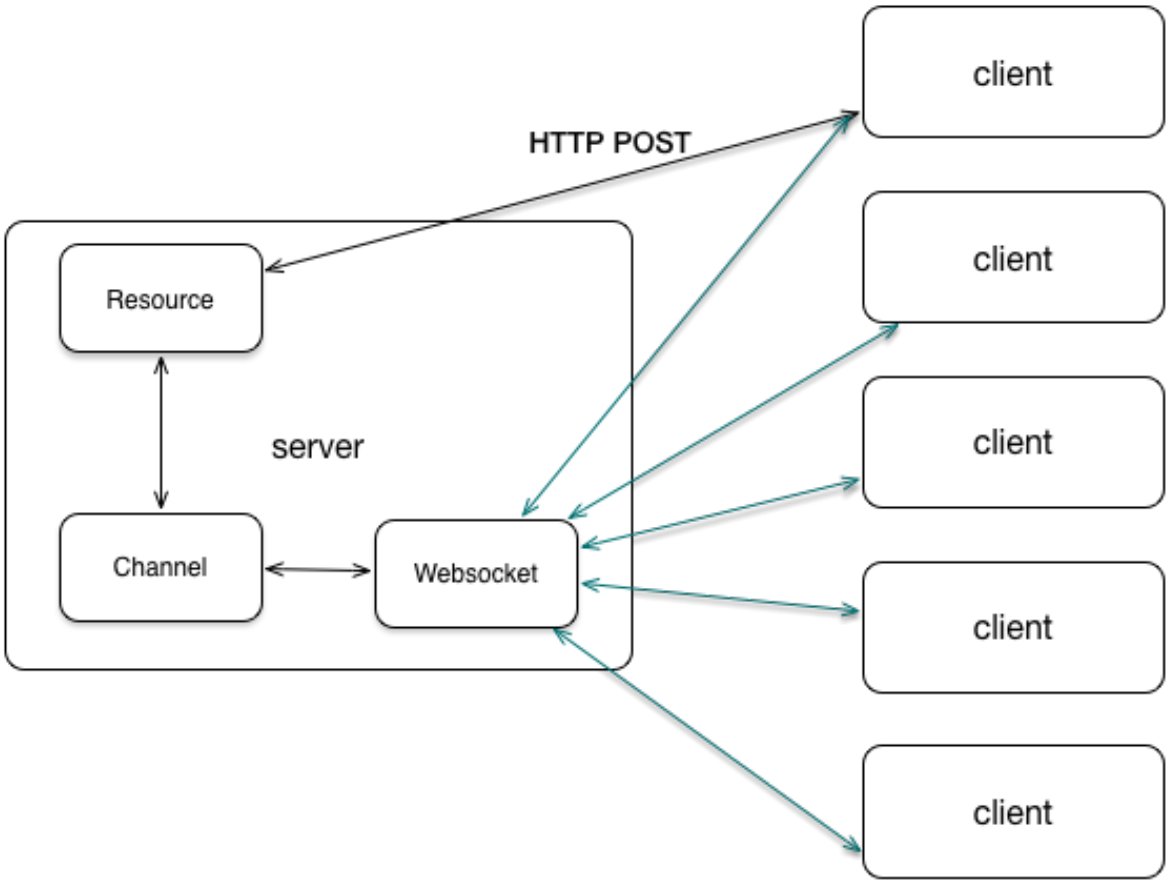
Parameters

- **`key`** – The name of the event
- **`data`** – The data to be passed with the event. The data must be such that it can be encoded to JSON

TBone uses the channels mechanism inside `Resource` based classes to implement full-duplex RESTful APIs. Therefore, resources can accept HTTP requests, but also send REST-like events. The `MongoResource` class uses a channel to publish resource events such as `created` or `updated` to implement a REST-like feedback on resource events. The scenario works like so:

1. A client send an http POST request to the resource, creating a new data object
2. The same resource class publishes an event to the channel that a new object was created, providing the serialized form of the object
3. The `Channel` object iterates through all subscribers (clients registered with a websocket connection) and sends the REST-like event to the registered clients

The following diagram illustrates this:



Channels are created as singletons based on the channel's name. This means that every channel given a name will have only a single instance running within the process. This is useful since channels can be created anywhere within the app components. By doing so, channels do not have to be injected into components.

Note: Channels are not restricted to usage by `Resource` objects. Any component can invoke a channel and send events.

1.7.2.1 Websockets

Creating a `Channel` and publishing events is not enough in order to send data to clients using websockets. Channels do not create the actual application endpoint which clients use to connect to the websocket interface. This has to be implemented by the developer, depending on the Webserver being used.

A minimal `Sanic` based example may look like this:

```
from tbone.dispatch.carriers.sanic_websocket import SanicWebSocketCarrier

async def resource_event_websocket(request, ws):
    # Create the channel - using the Mongo implementation
    request.app.pubsub = MongoChannel(name='pubsub', db=request.app.db)
    # Subscribe to the 'resource_create' event, passing the websocket instance,␣
↪wrapped in a Carrier subclass.
```

```
    request.app.pubsub.subscribe('resource_create', SanicWebSocketCarrier(ws))
    while True:
        await ws.recv()
    request.app.pubsub.unsubscribe('resource_create', SanicWebSocketCarrier(ws))
```

A minimal `AioHttp` based example may look like this:

```
from tbone.dispatch.carriers.aiohttp_websocket import AioHttpWebSocketCarrier

async def websocket_handler(request):

    ws = web.WebSocketResponse()
    await ws.prepare(request)

    # Create the channel - using the Mongo implementation
    request.app.pubsub = MongoChannel(name='pubsub', db=request.app.db)
    # Subscribe to the 'resource_create' event, passing the websocket instance,␣
↪wrapped in a Carrier subclass.
    request.app.pubsub.subscribe('resource_create', AioHttpWebSocketCarrier(ws))

    async for msg in ws:
        ...

    return ws
```

1.7.2.2 Carriers

Carriers are used by channels to abstract the mechanism in which events are sent through. Because TBone is webserver agnostic, supporting `AioHttp` websockets and `Sanic` websockets requires an abstraction layer over the websocket object itself. Furthermore, developers can subclass the `Carrier` class to implement additional mechanisms such as SockJS

1.8 Testing

1.9 Data

This section references the classes needed for defining data models. See the *Data Structure and Serialization* section for more detailed explanations

1.9.1 Fields

1.9.1.1 Base Field

1.9.1.2 Simple Fields

1.9.1.3 Composite Fields

1.9.1.4 Network Related Fields

1.9.1.5 MongoDB Fields

1.9.1.6 Extra Fields

1.9.2 Models

1.10 DB

This section covers the MongoDB persistency layer mixin and utility functions

1.10.1 MongoDB

1.11 Resources

This section covers API documentation for all classes and functions related to constructing REST API resources. For more information on resources in TBone see *Resources*

1.11.1 Authentication

class tbone.resources.authentication.**NoAuthentication**
Base class for all authentication methods. Used as the default for all resouces. This is a no-op authenttication class which always returns `True`

coroutine is_authenticated(*request*)
This method is executed by the `Resource` class before executing the request. If the result of this method is `False` the request will not be executed and the response will be 401 un authorized. The basic implementation is no-op and always returns `True`

1.11.2 Formatters

class tbone.resources.formatters.**Formatter**
Base class for all formatters. Subclass this to create custom formatters

format(*data: dict*)
Formats python `dict` into a data string. Implement in derived classes for specific transport protocols

parse(*body*)
Parses a string data to python `dict`. Implement in derived classes for specific transport protocols

class tbone.resources.formatters.**JSONFormatter**
Implements JSON formatting and parsing

1.11.3 Base Resource

class tbone.resources.resources.**ModelResource**(**args*, ***kwargs*)
A specialized resource class for using data models. Requires further implementation for data persistency

class tbone.resources.resources.**Resource**(**args*, ***kwargs*)
Base class for all resources.

class Protocol
An enumeration.

add_hypermedia(*obj*)
Adds HATEOAS links to the resource. Adds href link to self. Override in subclasses to include additional functionality

classmethod as_detail(*protocol=<Protocol.http: 10>*, **args*, ***kwargs*)
returns detail views

classmethod as_list(*protocol=<Protocol.http: 10>*, **args*, ***kwargs*)
returns list views

classmethod as_view(*endpoint*, *protocol*, **init_args*, ***init_kwargs*)
Used for hooking up the endpoints. Returns a wrapper function that creates a new instance of the resource class and calls the correct view method for it.

classmethod build_http_response(*data*, *status=200*)
Given some data, generates an HTTP response. If you're integrating with a new web framework, other than sanic or aiohttp, you **MUST** override this method within your subclass.

Parameters

- **data** (*string*) – The body of the response to send
- **status** (*integer*) – (Optional) The status code to respond with. Default is 200

Returns A response object

coroutine dispatch(**args*, ***kwargs*)
This method handles the actual request to the resource. It performs all the neccesary checks and then executes the relevant member method which is mapped to the method name. Handles authentication and de-serialization before calling the required method. Handles the serialization of the response

dispatch_error(*err*)
Handles the dispatch of errors

format(*method*, *data*)
Calls format on list or detail

classmethod nested_routes(*base_url*, *formatter: <built-in function callable>* = *None*) → list
Returns an array of Route objects which define additional routes on the resource. Implement in derived resources to add additional routes to the resource

Parameters

- **base_url** – The URL prefix which will be prepended to all nested routes
- **formatter** – The format method to be used when parsing url variables. By default the resource's route_param method is used, which formats the url based on which HTTP libary is used.

parse(*method*, *endpoint*, *body*)
calls parse on list or detail

request_args()
Returns the arguments passed with the request in a dictionary. Returns both URL resolved arguments and query string arguments. Implemented for specific http libraries in derived classes

coroutine request_body()
Returns the body of the current request. Implemented for specific http libraries in derived classes

request_method()
Returns the HTTP method for the current request.

classmethod route_methods()
Returns the relevant representation of allowed HTTP methods for a given route. Implemented on the http library resource sub-class to match the requirements of the HTTP library

classmethod route_param(*param*, *type=<class 'str'>*)
Returns the route representation of a url param, pertaining to the web library used. Implemented on the http library resource sub-class to match the requirements of the HTTP library

classmethod wrap_handler(*handler*, *protocol*, ***kwargs*)
Wrap a request handler with the matching protocol handler

class tbone.resources.resources.**ResourceOptions**(*meta=None*)
A configuration class for Resources. Provides all the defaults and allows overriding inside the resource's definition using the `Meta` class

Parameters

- **name** – Declare the resource's name. If `None` the class name will be used. Default is `None`
- **object_class** – Declare the class of the underlying data object. Used in `MongoResource` to bind the resource class to a `Model`
- **query** – Define a query which the resource will apply to all `list` calls. Used in `MongoResource` to apply a default query fiter. Useful for cases where the entire collection is never queried.
- **sort** – Define a sort directive which the resource will apply to GET requests without a unique identifier. Used in `MongoResource` to declare default sorting for collection.
- **hypermedia** – Specify if the a `Resource` should format data and include HATEOAS directives, specifically link to itself. Defaults to `True`
- **fts_operator** – Define the FTS (full text search) operator used in url parameters. Used in `MongoResource` to perform FTS on a collection. Default is set to *q*.
- **incoming_list** – Define the methods the resource allows access to without a primary key. These are incoming request methods made to the resource. Defaults to a full access `['get', 'post', 'put', 'patch', 'delete']`
- **incoming_detail** – Same as `incoming_list` but for requests which include a primary key
- **outgoing_list** – Define the resource events which will be emitted without a primary key. These are outgoing resource events which are emitted to subscribers. Defaults to these events `['created', 'updated', 'deleted']`
- **outgoing_detail** – Same as `outgoing_list` but for resource events which include a primary key
- **formatter** – Provides an instance to a formatting class the resource will be using when formatting and parsing data. The default is `JSONFormatter`. Developers can subclass `Formatter` base class and provide implementations to other formats.

- **authentication** – Provides and instance to the authentication class the resource will be using when authenticating requests. Default is `NoAuthentication`. Developers must subclass the `NoAuthentication` class to provide their own resource authentication, based on the application's authentication choices.
- **channel** – Defines the Channel class which the resource will emit events into. Defaults to in-memory

channel_class
alias of `Channel`

1.11.4 MongoDB Resource

1.11.5 AioHttp Mixin

1.11.6 Sanic Mixin

1.11.7 Router

1.12 Dispatch

This section covers the methods and classes used for handing internal and external events. For more information see *Dispatch*

1.12.1 Signals

class `tbone.dispatch.signals.`**Signal**
Base class for all signals

connect(*receiver*, *sender*)
Connects a signal to a receiver function

Parameters

- **receiver** – The callback function which will be connected to this signal
- **sender** – Specifies a particular sender to receive signals from. Used to limit the receiver function to signal from particular sender types

coroutine send(*sender*, ***kwargs*)
send a signal from the sender to all connected receivers

1.12.2 Channels

class `tbone.dispatch.channels.`**Channel**
Abstract base class for all Channel implementations. Provides pure virtual methods for subclass implementations

kickoff()
Initiates the channel and start listening to events. This method should be called at the startup sequence of the app, or as soon as events should be listened to. Pushes `consume_events` into the event loop.

coroutine publish(*key*, *data=None*)
Publish an event to the channel, to be sent to all subscribers

Parameters

- **key** – The name of the event
- **data** – The data to be passed with the event. The data must be such that it can be encoded to JSON

subscribe(*event*, *subscriber*)
Subscribe to channel events.

Parameters

- **event** – The name of the event to subscribe to. String based
- **subscriber** – A `Carrier` type object which delivers the message to its target

class `tbone.dispatch.channels.mem.`**MemoryChannel**(***kwargs*)
Represents a channel for event pub/sub based on in-memory queue. uses `asyncio.Queue` to manage events.

1.13 Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

W