Mojo Documentation

Release 0.1

Martin Buhr

January 14, 2014

Contents

1 Contents: 3

1.1 Quickstart: Using Mojo for the first time . . . 3
1.2 Mojo Apps . . . 5
1.3 Mojo Views . . . 5
1.4 Routing pages with URL's . . . 6
1.5 The Mojo ORM and Models . . . 7
1.6 Using Models . . . 7
1.7 Mojo and SocketIO . . . 10
1.8 Using Mojo's Auth and Session Mixins . . . 11

2 The Mojo API: 13

2.1 Mojo.Auth . . . 13
2.2 Mojo.Backends . . . 21
2.3 Mojo.ObjectMapper . . . 23
2.4 Mojo Form Helpers . . . 28
2.5 Mojo.RequestHandlers . . . 30
2.6 Mojo.SocketHandlers . . . 30

3 Indices and tables 33

Python Module Index 35

Mojo is a framework that makes it easy and quick to build Tornado projects that scale.

Some key features of Mojo:

- A lightweight and flexible ORM that makes developing easy
- ORM is based off a 'backend' system, enabling you to write your own backend and extend the ORM to other databases without altering your code
- Authentication and session mixins so you don't need to roll-your-own implementation or third parties
- Integration with wtForms, with the ability to use your Models as Forms and to populate models from Form data
- Modular structure so you can add functionality to your project on an app by app basis
- Prettier debugging output
- SocketIO support baked in with TornadIO2
- Project and app-creation templates that make it easy to set up new projects

The project is heavily influenced by Django, developers familiar with django will find some of the conventions in Mojo very familiar.

To get acquainted with Mojo and get started, see the quick-start guide.

CHAPTER 1

Contents:

1.1 Quickstart: Using Mojo for the first time

This is a quick-start tutorial that will get you set up and running with Mojo for the first time.

1.1.1 Installation

1. Download the Mojo distribution: https://raw.github.com/lonelycode/Mojo/master/dist/Mojo-0.1.tar.gz
2. Decompress the zip file
3. Make sure you have installed the required support packages: `Tornado`, `TornadIO2`, `bcrypt` and `wtforms`
4. From the command line in the new folder run `python setup.py install`

This should install Mojo into your python path. However, there is one more step that you may need to do to fully take advantage of Mojo's helper apps, and that is to make `mojo_manager` available in your `PATH`, for linux and Mac OSX users, this can be accomplished by doing something like:

```
ln /usr/bin/python2.7/Lib/site-packages/Mojo/mojo_manager.py /usr/sbin/mojo_manager.py
chmod +x /usr/sbin/mojo_manager.py
```

On windows, adding the Mojo site-packages directory should be enough to give gloabl access to `mojo_manager`

Once you've done that, you should be able to test your installation by opeining a python window and typing:

```
import Mojo
```

If there are no import errors then you shoulod be ready to get started.

Note: It is recommended to deploy Mojo in something like virtualenv to ensure you can easily keep it (and your app) up to date without affecting your main Python setup and other projects.

1.1.2 Your first project

Mojo sets up it's projects as a project folder that contains a series of Apps, these apps are independent from one another and offer a way to group different functional areas of your app away into discrete units. The typical Mojo project will have a folder structure that looks like:

```
--[ProjectName]
----[static]
----[Apps]
------[App 1]
--------[templates]
--------models.py
--------ui_modules.py
--------urls.py
--------views.py
------[App 2]
------[App 3]
----settings.py
----runserver.py
```

A quick run down of what each of these files and folders are:

- `[ProjectName]`: Your projct folder, this houses all the apps, modules settings and server for your tornado project
- `[static]`: All your static assets can be placed in here and they will be referenced automatically when you use the `static_url("images/logo.png")` built in tornado function
- `[Apps]`: Houses all of your individual apps, these break down into a series of base files that make your app work:
  - `[App 1]/models.py`: This is your model definition file, here you set out what database tables you want to use
  - `[App 1]/ui_modules.py`: Your UI Modules for Tornado are housed here, these are automatically loaded so they can be used directly from your templates
  - `[App 1]/urls.py`: The URL's for this app, ampping to the relevant Request Handlers
  - `[App 1]/views.py`: The request handlers that will manage the various app's functions
- `settings.py`: All the settings for your application
- `runserver.py`: This, strangely enough, runs your web server

To create your first app, you simply need to invok ethe mojo_manager application, this will create your project folder as follows:

```
> mojo_manager.py -p MyNewProject
> cd MyNewProject
> mojo_manager.py -a HelloWorldApp
```

That's it, all the files you need to get started should be created and in nbamed appropriately.

1.1.3 Setup the App

To get started, lets set up your settings.py to get your first server up and running. Open `settings.py` in your favourite editor and make sure the `INSTALLED_APPS` section looks like this:

```
INSTALLED_APPS = [
    'HelloWorldApp',
]
```

Once you've made the change, simply save the file and open up your terminal window in the directory where `runserver.py` is located, then type the following:

```
python runserver.py
```

You should see:

```
Starting Mojo tornado server.
DEBUG:root:Setting up url routers:
DEBUG:root:--Added URL's for: blog_app
DEBUG:root:--Adding UI Modules for blog_app
DEBUG:root:--Added SocketHandler for: blog_app
DEBUG:root:Found DATABASE setting - creating session for DB: mojo_blog
INFO:root:Starting up tornadio server on port '8001'
INFO:root:Entering IOLoop...
```

If you navigate to `http://localhost:8000` you should see the Mojo welcome page. *Congratulations, you are running Mojo!*

1.2 Mojo Apps

When developing with Mojo, it's more elegant to modularise functionality into Apps, and this is the default behaviour of the Mojo server loader.

Apps serve as a collection of RequestHandlers, URL Mappings, Models and Templates that make up a set of functionality in your project.

To get started with an app in your project, you can use the `mojo_manager.py` helper application that will auto create your project folder for you:

```
python mojo_manager.py -a [YOUR_APP_NAME]
```

This will create an app folder with certain key files:

- `models.py`: This is where you define your models which can be used with your database and forms
- `socket_handlers.py`: This is where you define your SocketIO behaviour, each app has it's own dedicated (multiplexed) channel under `/[appname]Socket` to separate out functionality
- `ui_modules.py`: The ui_modules you might or mihjgt not be using
- `urls.py`: The URL mappings for your request handlers
- `views.py`: A list of your `RequestHandler` classes that handle your app functionality

1.3 Mojo Views

Mojo views are where you will write most of your application and page handling logic. In the `views.py` file, you will sub-clas the MojoRequestHandler class to create your application.

The MojoRequestHandler is a slightly modified version of the basic Tornado RequestHandler that makes the local templates available to the interpreter and adds support for prettier error formatting, it is also required to make the `SessionMixin` and `AuthMixin` classes to work as they depend on the `MojoRequestHandler` base class.

Note: It completely possible to develop with Mojo using standard `equestHandlers`, simply subclass them as you would normally and use them in your `urls.py`

1.3.1 Views Quick-start:

To write your first view, subclas the MojoRequestHandler class, type thi into your view handler:

```
from Mojo.RequestHandlers.MojoHandler import MojoRequestHandler

class HelloWorldHandler(MojoRequestHandler):
    def get(self, *args, **kwargs):
        self.render('hello.html')
```

And then mak sure it is accessible in your `urls.py` file:

```
from views import *

urlpatterns = [
    #Place your URL Routes / RequestHandler mappings in here for this app, e.g.
    ('/',      HelloWorldHandler),
]
```

You will also need to actually create the template file, `hello.html`, it could lok somwthing like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Hello from Mojo!</title>
</head>
<body>
    <h1>Hi there, I'm a template inside Mojo!</h1>
    <p>This is a tempalte file that is stored in your apps template directory.</p>
</body>
</html>
```

Save this file to your app `templates` directory and make sure that your app is listed in the `settings.py` `INSTALLED_APPS`
setting:

```
INSTALLED_APPS = [
     'HelloWorldApp',
]
```

Now you just need to run your server, in a command line window, in your project directory type:

```
python runserver.py
```

If your server is already running, and you have `DEBUG=True` set in your `settings.py` file, it should automatically restart when you save the above changes.

When you navigate to `http://localhost:8000/` you should see your template being rendered out...

The MojoRequestHandler behaves the same way as the Tornado request handler, we recommend the reader check the Tornado documentation to get fully up to speed with what the capabilities are of `RequestHandler` objects.

1.4 Routing pages with URL's

Mojo moves the URL definition of your app to the file `urls.py`, this file contains a list object called `url_patterns` that contains tuples mapping a URL to a `RequestHandler` in your `views.py`.

All entries work the same way as they do with standard Tornado mappings, when your server starts up, Mojo will automatically concatenate all the URL's for you into a single pattern set to ensure that all your apps are available in the project without mixing functionality across apps:

```
from views import myRequestHandler

urlpatterns = [
    ('/hello', myRequestHandler),
    ]
```

All URL patterns can be regex groups - see the Tornado documentation for more details on how to use the URL routing in a Tornado app.

1.5 The Mojo ORM and Models

To make working with databases easier, and to ensure a consistent set of tools when saving and moving data, Mojo uses models to define validation and data structures, and then uses these to set up how data will look in the database.

Currently, Mojo only supports MongoDB, but we are hoping to add support for Redis and CouchDB soon. We actively encourage contributors, so anyone feeling the urge to help us write a new backend for their favourite datastore can drop us a line in our Google Group.

1.5.1 What are Models?

For anyone familiar with Django, Mojo Models will seem very similar, and take heavy influence from the framework. Models are a way for you to organise and structure your data as python objects so you can transparently use them (without a data store) in your application.

Models are very simple - they are a defined class, with a series of fields that you can use to represent your data and the various validation methods you would like run against those fields before the object is written to the database.

1.5.2 Why Models?

Because our first database to support was MongoDB, it became apparent that what was really missing was a good way to enforce structure and a certain degree of validation on stored data. Being Schema-less, it becomes very easy to not validate stored input due to the extra burden of writing the validation code, as well as not manging schema evolutions as schemas change over the lifecycle of your project.

Models make this process easiewr by pre-defining data types, relations and object properties prior to saving, while exposing all the benefits of having a neat python object to represent your data set without resporting to direct database access.

1.5.3 Do I have to use Models?

Not at all, you can completely ignore the ORM and model structure in your project and directly access your database using your favourite driver.

1.6 Using Models

To implement a model for your app, simply define it in the `models.py` file in your app directory:

```
from Mojo.ObjectMapper.ModelPrototype import Model
from Mojo.ObjectMapper.Fields import *
import datetime

#This is a non-database class, not stored in the DB but embedded in the BlogPost class below
#the key difference is the lack of an _id field
class Tag(Model):
    tag_name = StringField()

#a simple model to hold blog posts
class BlogPost(Model):
    _id = ObjectIDField()
    title = StringField(allow_empty=False)
    slug = StringField(allow_empty=False)
    post_intro = StringField()
    post_body = StringField()
    date_published = DateTimeField(default=datetime.datetime.now())
    tags = ListField(of=Tag)
    visible = BooleanField(default=True)
```

In the above example we've defined two models, the first `Tag` model is a straightforward class that is **not** a database object the key distinction here is the lack of the `_id` field (this is only relevant to the MongoDB backend as it requires an ID to be explicitly defined).

The second model `BlogPost` is the real model, and lists a series of fields to represent data types, fields can be anything, so long as they subclass the `Mojo.ObjectMapper.FieldPrototype.Field` object.

You can access the above model quite easily in your code:

```
new_minimal_post = BlogPost({
    'title':u'a new post',
    'slug' : 'newpost'
})

#access some properties
print new_minimal_post.title
> a new post

#save it (assuming synchronous DB backend):
thispost = new_minimal_post.save()

print thispost._id
> 50571e5f3d941cdc4487bdf1
```

Models can be assigned data in dictionary format as part of their initialisation, or empty and then using dot-notation for each of their properties. So, in the example above, the following would work just as well:

```
new_minimal_post = BlogPost()
new_minimal_post.title = u'a new post'
new_minimal_post.slug = 'newpost'

#access some properties
print new_minimal_post.title
> a new post

#save it (assuming synchronous DB backend):
thispost = new_minimal_post.save()

print thispost._id
```

```
> 50571e5f3d941cdc4487bdf1
```

`Model` objects are dictionaries, so if you print out or access them in any way, they should behave in the same way as a stamdard python `dict` object.

Saving is very straightforward in mojo, and as can be seen from the example above, can easily be accomplished with the `save()` instance method.

The main read/write operations of a model are as follows:

- `save()`: Called on an instance of the model, this will attempt to save the data to the database, if the object has an initialised `_id` value then it will be updated, otherwise it will perform an insert
- `delete()`: Called on an instance of the model, thjis will delete it from the database
- `find()`: Called on the Model object (class method), will use the criteria passed to find to return a list of instantiated model instances
- `find_one()`: Called on the model object (class method), will use the criteria passed to find a single model object
- `delete_bulk([list])`: Pass a list of objects ot this function for a bulk delete operation

Both the `find()` and `find_one()` operations take MongoDB style `dict` objects as request parameters and follow the pymongo style of queries.

One of the most appealing aspects of Tornado is it's ability to work asynchronously, and Mojo takes that to heart, ensuring taht you can just as well use the asynchronous style of development with the ORM and Models.

To use the Asynchronous driver, make sure that you have changed it in your `settings.py` file:

```
DATABASE = {
    'backend': 'Mojo.Backends.AsyncmongoBackend.asyncmongo_backend',
    'name': 'YOUR_DB_NAME',
    'host': '127.0.0.1',
    'port': 27017
}
```

To start using it in your code, all models make an `_async` version of all operations available, that can be used with traditional callback-style async handling or Tornado's `gen` module style (for more readable code):

```
class myRequestHandler(MojoRequestHandler):
    @gen.engine
    @tornado.web.asynchronous
    def get(self):
        from bson.objectid import ObjectId
        thisPost = yield gen.Task(BlogPost.find_one_async, {'_id':ObjectId('5059fb6b3d941cdc4487bdff'

        self.render('template.html', post=thisPost)
```

If you want to use callback-style development:

```
class myRequestHandler(MojoRequestHandler):

    def callback(value):
        self.render('template.html', post=value)

    @gen.engine
    @tornado.web.asynchronous
    def get(self):
        from bson.objectid import ObjectId
        yield gen.Task(BlogPost.find_one_async, {'_id':ObjectId('5059fb6b3d941cdc4487bdff')}, callba
```

NOTE: The ORM and Models modules are still in very early stage development, please report any bugs to the developer on the source-control page on BitBucket.

1.7 Mojo and SocketIO

One of the strengths of Tornado is it's ability to handle asynchronous requests and support for websockets. Top make Mojo more of a responsive framework, we've decided to bundle `TornadIO2` functionality as part of the overall package.

For those that do not know what `TornadIO2` is, it is a `SocketIO` server implementation written in Python for Tornado, and makes it transparent to implement SocketIO clients into your app.

By default, Mojo will start a `TornadIO` server for you that multiplexes a channel connection for each of the apps in your project.

So if you have 2 apps: `App1` and `App2`, then you will be able to speak to either of these apps via SocketIO on the client by connecting to the `/App1Socket` or `/App2Socket` channels in your client side configuration.

To configure what commands the server should intitiate on connection, send event or receive event, simply edit the `socket_handlers.py` file in your app folder:

```
from Mojo.SocketHandlers.BaseSocketHandler import MojoSocketHandler, CURRENT_SESSIONS, LOGGED_IN_SESS

#Setup your socket connections here
class SocketConnectionHandler(MojoSocketHandler):

    def on_message(self, msg):
        #Do something cool when a message arrives
        pass
```

To make the most of `TornadIO2` and `SocketIO` we'd recommend you read the documentation for both as their inner workings fall outside of the remit of the Mojo documentation.

1.7.1 Helpful Tracking

Mojo does some handy tracking for you if you use the `MojoSocketHandler` to manage your socket connections. Mojo will automatically register all connections in two variables that enable you to communicate with users directly from the start, these variables are `CURRENT_SESSIONS` and `LOGGED_IN_SESSIONS`

CURRENT_SESSIONS

This is a dictionary object that will track each connection by their Session ID. The session ID is a secure Tornado cookie that is called `session_id`, if you are using the `MojoSessionMixin` module, this will be set for you, however if you want to roll your own session management, you can just set this cookie and the user will be tracked by the ID in the `SocketHandler`.

This will enable you to do something like:

```
from Mojo.RequestHandlers.MojoHandler import MojoRequestHandler
from Mojo.SocketHandlers.BaseSocketHandler import CURRENT_SESSIONS
import json

    class HelloWorldHandler(MojoRequestHandler):
        def get(self, *args, **kwargs):
            jsonObj = json.dumps({'data':'%s has joined the chat' % (self.session_id)})

            for session in CURRENT_SESSIONS:
                session.emit('joinedStatus', jsonObj)

            self.render('hello.html')
```

Although the above code is not functional, the principle is valid - in this example, you could use the CURRENT_SESSIONS object to broadcast to all active users that a new user has joined the conversation.

`LOGGED_IN_SESSIONS`

If you are using the AuthMixin from Mojo, then it will set an encrypted cookie of the name `logged_in` with the user ID of the current user. Naturally, you can set this yourself to whatever identifier you like if you decide not use the mixins.

Any session that is identified as Logged in will be added to the `LOGGED_IN_SESSIONS` dictionary, and will enable you to interact with users. In a similar vein to the `CURRENT_SESSIONS` object, you would now be able to send specific messages only to logged in members of your app.

1.8 Using Mojo's Auth and Session Mixins

To make Tornado a bit easier to use and integrate with your application, Mojo provides a sesison management mixin that adds database-backed sessions to your requests.

The Session Mixin assumes that a database backend and ORM is present, which is why it comes in both sync and async flavors. The AuthMixin does not assume that a database is present and solely uses the set_cookie functionality of Tornado to make the get_current_user functionality work.

1.8.1 AuthMixin

The AuthMixin basically overrides get_current_user() in the MojoRequestHandler and adds a SessionManager object to the attributes of the RequestHandler. Using the SessionManager, it is possible to login/logout the user:

```
if thisUser is not None:
    if is_authenticated:
        SessionManager(request)._login(thisUser) #Will make the relevant changes to the cookies.
        return True
    else:
        return False
else:
    return False
```

Using it in a request might look like:

```
import tornado.web
from Mojo.RequestHandlers.MojoHandler import MojoRequestHandler
from Mojo.Auth.Mixins.MojoAuthMixin import MojoAuthMixin
from Mojo.Auth.models import User
from Mojo.Auth.Helpers import login_assistant

#To implement the mixin, simply subclass it alongside the regular MojoRequestHandler and the authent
#funcitonality will be come available.
```

```
class loginHandler(MojoRequestHandler, MojoAuthMixin):

    def get(self):
        #Override the get function to render the page, check current_user to see if we're already log
        if self.current_user:
            self.render('login.html', error='ALREADY LOGGED IN')
        else:
            self.render('login.html', error=None)

    def post(self):
        #Get the username and password from the request
        username = self.get_argument('username')
        password = self.get_argument('password')

        target = self.get_argument('next', '/admin/')

        #Get the user from the database
        thisUser = User.find_one({'username':username})

        #Log the user in using the login assistant
        if login_assistant(thisUser, password, self):
            self.redirect(target)
        else:
            self.render('login.html', error='Login failed')
```

1.8.2 The SessionMixin

The SessionMixin comes in two flavours: Syncronous and Asyncronous, depending on what backend is being used in Mojo. To implement the SessionMixin, simply add it to the inheritance list as part of your RequestManager:

```
from Mojo.Auth.Mixins.SessionMixins import SessionMixin_Sync
class loginHandler(MojoRequestHandler, SessionMixin_Sync):

            def get(self):
                pass
```

The SessionMixin exposes some basic functionality that will let you get and set session data as part of your site and save it to the database.

All session data is stored in the SessionModel as a Base64 encoded string that is a `dict`. To get and set session values is quite straightforward as the session will be auto-created as soon as you start using the object:

```
from Mojo.Auth.Mixins.SessionMixins import SessionMixin_Sync
    class loginHandler(MojoRequestHandler, SessionMixin_Sync):

            def get(self):
                self.set_session_key('keyname', 'value')

                value = self.get_session_key('keyname')

                self.render('template.html', session_key_value = value)
```

CHAPTER 2

The Mojo API:

2.1 Mojo.Auth

The Mojo.Auth modules are designed to make authentication easier in Tornado by providing a model framework, password management, encryption and validation tools.

To accompany the authentication tools, there is also a session management system that makes maintaining persistent sessions easy across requests.

This module also provides a set of Mixins that integrate the authentication and session management functionality and integrate with either the synchronous or asynchronous mongo drivers, see the mixins section to see how this is accomplished.

2.1.1 AuthManager

`Mojo.Auth.AuthManager.`**`add_to_group`**(*userObj*, *groupObj*)

Will add a group object to the groups list in the User object. returns a `Mojo.Auth.models.User` object.

`Mojo.Auth.AuthManager.`**`authenticate`**(*userObj*, *password*)

Will check if a user password matches the input to authenticate the user, this will **not** log them in:

```
from Mojo.Auth.AuthManager import authenticate

thisUser = Users.find_one({'_id':ObjectId('504e0439a9ee2f04a0835a92'})
authenticated = authenticate(thisUser, password)
if authenticated:
    #Process the login
    ...
else:
    ...
```

Will return `True` if passwords matches and `False` if password doesn't match.

Authentication in Mojo uses the `bcrypt` library for encrypting passwords and is a requirement for using the Auth module.

`Mojo.Auth.AuthManager.`**`is_member_of`**(*userObj*, *groupObj*)

Will determine is a user is a member of a group. Returns `Tru` or `False`

`Mojo.Auth.AuthManager.`**`login`**`(`*userObj*`)`

If you are using the Auth module and want to track login times as part of the userObj, use this to set the appropriate state of the user objects last_login field, can be used in conjunction with `Mojo.Auth.Helpers.login_assistant()` to round-off the login process:

```
from Mojo.Auth.AuthManager import login
from Mojo.Auth.Helpers import login_assitant

thisUser = Users.find_one({'_id':ObjectId('504e0439a9ee2f04a0835a92'})

if thisUser:
    is_authenticated = login_assistant(thisUser, password, self) #self in this case is a Request
    if is_authenticated:
        login(thisUser)              #set the last_login
        thisUser.save()              #save the user
    else:
        ...
```

`Mojo.Auth.AuthManager.`**`make_random_password`**`()`

Generates a random string consisting of 8 characters chosen from the uppercase ascii alphabet and digits.

Returns string.

`Mojo.Auth.AuthManager.`**`remove_from_group`**`(`*userObj*, *groupObj*`)`

Removes a user from the group, returns `True` if successful, `False` if the user is not a member of the group

`Mojo.Auth.AuthManager.`**`set_password`**`(`*userObj*, *new_password*`)`

Sets the user password and returns the user object, *the object is not saved to the database*. Usage requires `bcrypt` to be installed alongside Mojo:

```
from Mojo.Auth.AuthManager import set_password

thisUser = Users.find_one({'_id':ObjectId('504e0439a9ee2f04a0835a92'})
thisUser = set_password(thisUser, 'new-password=string')
thisUser.save() #save the user object to the database.
```

2.1.2 Helpers

`Mojo.Auth.Helpers.`**`login_assistant`**`(`*thisUser*, *password*, *request*`)`

Shortcut function that will perform three things at once:

1. Checks if the user exists
2. Authenticates the user against a password using `Auth.AuthManager.authenticate()`
3. If authenticated, sets the appropriate session cookie (that is checked by `get_current_user()` and returns `True`

Parameters

- `thisUser` - Mojo.Auth.models.User object
- `password` - string password representation
- `request` - RequestHandler object, does *not* require the Session Mixin to be used.

Usage:

```
from Mojo.Auth.Helpers import login_assitant

thisUser = Users.find_one({'_id':ObjectId('504e0439a9ee2f04a0835a92'})
if thisUser:
    is_authenticated = login_assistant(thisUser, password, self) #self in this case is a Request
    if is_authenticated:
        ...
    else:
        ...
```

`Mojo.Auth.Helpers.`**`logout_assistant`**(*request*)

Shortcut function to log a user out. Uses `Mojo.Auth.SessionManager` to set the appropriate session cookie and returns `True`.

Once called, `get_current_user()` and `curent_user` will be returned as `None` in a RequestHandler.

2.1.3 Auth Models

class `Mojo.Auth.models.`**`Group`**(*data=None*)

Group definition - consists only of the name of the group, is part of a `ListField` in the `User` object

class `Mojo.Auth.models.`**`Profile`**(*data=None*)

Represents a user profile - currently cannot be extended. Is an embedded document in a `User` object.

class `Mojo.Auth.models.`**`Session`**(*data=None*)

The `Session` object is mainly used as part of the `SessionManager` class and the `SessionMixin` mix-in base class.

This should **not** be manipulated directly, but stores any session information in the `session_data` database field.

Data stored in `session_data` is a JSON string that is base64 encoded, user log in flags are *not* stored in `session_data` to avoid hitting the database every time `current_user` is queried in a request.

class `Mojo.Auth.models.`**`User`**(*data=None*)

Main User model used by the `Mojo.Auth` modules, consists of multiple fields:

- `username`: The username of the user (required)
- `password`: The `bcrypt` encrypted password (required)
- `email`: the email address of the user (optional)
- `groups`: A list field of `Group` objects that the user belongs to
- `active`: Is the user active, and allowed in?
- `profile`: An embedded field that uses the `Profile` model
- `is_superuser`: Flag to indicate super user status (`BooleanField`)
- `last_login`: `DateTimeField` with last login date/time
- `date_joined`: Can be set to when the user joined (`DateTimeField`)

2.1.4 Session Manager

`Mojo.Auth.SessionManager.`**`Reset_session`**(*sessionObj*, *expiry_days=30*, *expiry_hours=0*, *expiry_minutes=0*)

Resets an exisitng session - for example, if a user has logged in again, the expiry date needs to be extended to another 30 days. Takes the same parameters as `Setup_session()`:

Paramaters:

- `sessionObj` - The session_model to be initialised
- `expiry_days` - How long the session shoul last (optional, default is 30 days)
- `expiry_hours` - How long the session should last i hours (optional)
- `expiry_minutes` - How long the session should last in minutes (optional)

class `Mojo.Auth.SessionManager.`**`SessionManager`**(*request_handler*)

`SessionManager` is meant to make it easier to manage sesison cookies and persistent session data by providing a series of methods that can be used to log a user in, log a user out, get all cookie data.

The `SessionManager` also manages the encoding, decoding, getting and setting of persistent session data and session valididty, it can be used on it's own or through the `SessionMixin` base class.

For example, the `Mojo.Auth.Helpers.login_assitant` function will use a `SessionManager` class to get and set the cookies that log a uder in and out, while the `SessionMixin` class will also get and set persistent session data and save it to the database automatically, making sesison manageent easier and more secure.

`SessionManager` objects take a `RequestHandler` as an init parameter in order to function:

```
if thisUser is not None:
    if is_authenticated:
        SessionManager(request)._login(thisUser) #Will make the relevant changes to the cookies.
        return True
    else:
        return False
else:
    return False
```

`_create_new_session`(*callback=None*)

Creates and sets up a new `session_model` ready for the request.

`_decode_session`(*session_data*)

Internal helper function that decodes the `session_data` attribute of the `session_model`.

`_encode_session`(*session_data*)

Internal helper function that encodes the `session_data` attribute of the `session_model`.

`_get_session_key`(*key*)

Gets a session value stored in the session model by it's key. Session data is stored as a base64-encoded `JSON` string, and so the function will go through a *check* -> *decode* -> *set* -> *encode* -> *return* process and get the `key` value of the `session_data` of the `session_model` attribute.

`_is_logged_in`()

Checks if a user is logged in, will get the session cookies and check the `logged_in` value to see if it is None returns `True` or `False` depending on the outcome.

`_is_session_valid`()

This queries the session model in the database to check if the session has expired or not - this is *not* the same as the cookie expiry. Returns `True` or `False`

`_login`(*userObj*)

Set the secure cookie to log a user in, the valud of `logged_in` is set to the `_id` of the User object.

`_logout`()

Set the secure cookie to log a user out - essentially setting the `logged_in` session cookie to blank.

`_set_session_key`(*key*, *value*)

Sets a session key to be stored in the session model. Session data is stored as a base64-encoded `JSON` string, and so the function will go through a *check* -> *decode* -> *set* process and set the `session_model` attribute of the `SesionManager`

get_session_cookies()

Returns Mojo-specific cookie data:

- `session_id` - the ID used to lookup the session in the database
- `logged_in` - the _id of the User model that is currently logged in

Cookie data is returned as a `dict`

`Mojo.Auth.SessionManager.`**Setup_session**(*sessionObj*, *expiry_days=30*, *expiry_hours=0*, *expiry_minutes=0*)

Helper function that will take a ''session_model" object and apply the base data to it ready for use in a ''RequestHandler" or ''SessionManager'‘. By default, a session will expire after 30 days.

Paramaters:

- `sessionObj` - The session_model to be initialised
- `expiry_days` - How long the session shoul last (optional, default is 30 days)
- `expiry_hours` - How long the session should last i hours (optional)
- `expiry_minutes` - How long the session should last in minutes (optional)

Usage:

```
from Auth.SessionManager import Setup_session
from Auth.models import Session

new_session = Setup_session(Session())
```

2.1.5 Mojo.Auth.Mixins

Mixin Modules to help with Authentication and Session management

Mojo.Auth.Mixins

The Mojo Mixins are there to make authentication and session managemeent easy in your application without having to load in the whole machinery in each request.

MojoAuthMixin

MojoAuthMixin is designed to override the get_current_user functionality of a standard Tornado RequestHandler to provide straightforward and secure authentication. Makes use of the SessionManager class to manage cookies.

class `Mojo.Auth.Mixins.MojoAuthMixin.`**MojoAuthMixin**(*application*, *request*, ***kwargs*)

The Auth mixin will make the get_current_user functionality available that takes advantage of the Mojo.Auth familly of modules and models.

Example Usage:

```
import tornado.web
from Mojo.RequestHandlers.MojoHandler import MojoRequestHandler
from Mojo.Auth.Mixins.MojoAuthMixin import MojoAuthMixin
from Mojo.Auth.models import User
from Mojo.Auth.Helpers import login_assistant

#To implement the mixin, simply subclass it alongside the regular MojoRequestHandler and the aut
#funcitonality will be come available.
```

```
class loginHandler(MojoRequestHandler, MojoAuthMixin):

    def get(self):
        #Override the get function to render the page, check current_user to see if we're alreac
        if self.current_user:
            self.render('login.html', error='ALREADY LOGGED IN')
        else:
            self.render('login.html', error=None)

    def post(self):
        #Get the username and password from the request
        username = self.get_argument('username')
        password = self.get_argument('password')

        target = self.get_argument('next', '/admin/')

        #Get the user from the database
        thisUser = User.find_one({'username':username})

        #Log the user in using the login assistant
        if login_assistant(thisUser, password, self):
            self.redirect(target)
        else:
            self.render('login.html', error='Login failed')
```

get_current_user()

Overrides `get_current_user` to return the `logged_in` value from the sesison cookies. This function uses the `SessionManager` class to get and set cookies (this is to ensure that Mojo-specific functionality and keys are consistent).

SessionMixins

Session mixins are designed to make persistent session management features from SessionManager available as part of your RequestHandler, the mixins enable getting and setting of persistent session data (sessions are stored in the database) and also nifty helper functions that wrap SessionManager.

The mixins come in two flavours: Synchronous and Asynchronous to ensure they work fully with your preferred database backend.

Synchronous (blocking) Session Mixin

class `Mojo.Auth.Mixins.SessionMixins.`**SessionMixin_Sync**(*application*, *request*, ***kwargs*)

Synchronous Session Mixin `RequestHandler` base class. Exposes session management functions via a `SessionManager` object and ties these back using the ORM to the database, this mixin will use a blocking driver.

Usage:

```
from Mojo.Auth.Mixins.SessionMixins import SessionMixin_Sync

class loginHandler(MojoRequestHandler, SessionMixin_Sync):
    def get(self):
        ...
```

create_new_session()

Wrapper around the SessionManagers _create_new_session() method, but will save the session to DB instead of having to manage it manually.

get_session_key (*key*, *default=None*)

Gets a session key from the database based on the `session_id` supplied by the `RequestHandler`. Similarly to `set_session_key`, this is not a cookie value, but a persistent session variable from the database.

Usage:

```
class loginHandler(MojoRequestHandler, SessionMixin_Sync):

    def get(self):
        this_session_value = self.get_session_key('test_value')

        #Should render the 'test_value' session variable if it's in the template.
        self.render('main.html', session_value=new_value)
```

get_session_object ()

Returns the whole session_model object and assigns it to itself.

get_user_from_db (*uid=None*, *username=None*)

Gets a user from the database, this is such a common operation it offers a quick and simple way to return the full user object from the database either by supplying the `username` or `password`.

save_session_object ()

Saves the session model to the database, in this case it's a synchronous (blocking) operation. If there is no session to save, will create a new one (which is then saved automatically)

session

Session property - holds a `SessionManager` object that is initialised with the current `RequestHandler` as context, will initialise on first access.

set_session_key (*key*, *value*)

Sets a session key and saves it to the database (not a cookie - sessions are identified by a `session_id` in the secure cookie collection and for security purposes are encoded and stored in the database so as not to leak any information).

Usage:

```
class loginHandler(MojoRequestHandler, SessionMixin_Sync):

    def get(self):
        self.set_session_key('test_value', 'hello world!')
        new_value = self.get_session_key('test_value')

        #Should render the 'test_value' session variable if it's in the template.
        self.render('main.html', session_value=new_value)
```

Asynchronous (non-blocking) Session Mixin

class `Mojo.Auth.Mixins.SessionMixins.`**SessionMixin_Async** (*application*, *request*, ***kwargs*)

Asynchronous Session Mixin `RequestHandler` base class. Exposes session management functions via a `SessionManager` object and ties these back using the ORM to the database, this mixin will use a non-blocking driver.

Is compatible with gen.Task or callback-style implementations, the preferred method is the gen.Task implementation.

Usage:

```
from Mojo.Auth.Mixins.SessionMixins import SessionMixin_Async

class loginHandler(MojoRequestHandler, SessionMixin_Async):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        ...
```

create_new_session(**args*, ***kwargs*)
Wrapper around the SessionManagers _create_new_session() method, but will save the session to DB instead of having to manage it manually.

get_session_key(**args*, ***kwargs*)
Gets a session key from the database based on the `session_id` supplied by the `RequestHandler`. Similarly to `set_session_key`, this is not a cookie value, but a persistent session variable from the database.

Usage:

```
class loginHandler(MojoRequestHandler, SessionMixin_Sync):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        new_value = yield gen.Task(self.get_session_key,'test_value')

        #Should render the 'test_value' session variable if it's in the template.
        self.render('main.html', session_value=new_value)
```

get_session_object(**args*, ***kwargs*)
Returns the whole session_model object and assigns it to itself.

get_user_from_db(*uid=None*, *username=None*)
Gets a user from the database, this is such a common operation it offers a quick and simple way to return the full user object from the database either by supplying the `username` or `password`.

save_session_object(**args*, ***kwargs*)
Saves the session model to the database, in this case it's an asynchronous (non-blocking) operation. If there is no session to save, will create a new one (which is then saved automatically)

session
Session property - holds a `SessionManager` object that is initialised with the current `RequestHandler` as context, will initialise on first access.

set_session_key(**args*, ***kwargs*)
Sets a session key and saves it to the database (not a cookie - sessions are identified by a `session_id` in the secure cookie collection and for security purposes are encoded and stored in the database so as not to leak any information).

Usage:

```
class loginHandler(MojoRequestHandler, SessionMixin_Async):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        yield gen.Task(self.set_session_key,'test_value', 'hello world!')

        new_value = yield gen.Task(self.get_session_key,'test_value')
```

```
            #Should render the 'test_value' session variable if it's in the template.
            self.render('main.html', session_value=new_value)
```

2.2 Mojo.Backends

Mojo abstracts the database interaction away from the object mapper through the use of a backend. These backends offer a 'safe' interface for the model prototype to use to access base functions of the driver (such as `save`, `find`, `find_one` etc.).

The back-ends essentially treanslate the input parameters down to the base driver. Currently Mojo ships with two backends: one for `Pymongo` and the other for `Asyncmongo`.

Part of the reason for developing Mojo was to be able to easily decide what style of driver to use without having to rewrite many functions (or at least with only minor changes), as the asynchronous style of writing is so different from traditional development while retaining data integrity in your schemas.

Mojo supports both blocking and non-blocking drivers to ensure that both styles of development can be harnessed without forcing the developer down a specific path.

It is easy to add new back-ends by subclassing the `Mojo.Backends.base_interface` and referencing it in your settings file.

class `Mojo.Backends.base_interface.`**`CollectionModelInterface`**(*session*, *model*)

The CollectionModelInterface exposes the basic functions of the driver to the ModelPrototype, it will expect to have these functions *at least* in order to be fully functional:

- `find`
- `find_one`
- `save`
- `delete`

To create your own back-end, subclass this class and override all the functions with the relevant access functions and parameter styles.

Mojo tends to use Pymongo style dictionary access, it is recommended to try to adhere to this format.

`delete`(*documents*, **args*, ***kwargs*)
Override this to enable `delete` in the database

`find`(**args*, ***kwargs*)
Override this to enable `find` in the database

`find_one`(**args*, ***kwargs*)
Override this to enable `find-one` in the database

`insert`(*documents*, **args*, ***kwargs*)
Override this to enable `insert` in the database

`save`(*document*, **args*, ***kwargs*)
Override this to enable `save` in the database

class `Mojo.Backends.base_interface.`**`SessionInterface`**(*host='127.0.0.1'*, *port=27017*, *db_name=None*)

Session wrapper around the database connection, takes host, port and database information to enable the database session.

If your database has a specific style of connecting, subclass this class and override the `_setup_connection` function to connect to make the connection available to your backend.

Back-ends access the Session through a `session._db` property that is assigned during the server boot up:

```
#Snippet from the Pymongo backend

def find_one(self, *args, **kwargs):
    return_dict = self.session._db[self.collection_name].find_one(*args, **kwargs)
    return self._return_model_object(return_dict)
```

2.2.1 Included Backends

Currently Mojo ships with two backends:

- `Pymongo` (blocking)
- `Asyncmongo` (non-blocking)

Using Backends

To use a backend in your project, make sure you have the driver installed using pip or setup tools first, and then add the following to your `settings.py` file:

```
DATABASE = {
    'backend': 'Mojo.Backends.<BACKEND-MODULE>.<BACKEND>',
    'is_async': False,
    'name': 'test',
    'host': '127.0.0.1',
    'port': 27017
}
```

for example, to use the asyncmongo back-end:

```
DATABASE = {
    'backend': 'Mojo.Backends.AsyncmongoBackend.asyncmongo_backend',
    'is_async': False,
    'name': 'test',
    'host': '127.0.0.1',
    'port': 27017
}
```

Any models you implement will use the appropriate driver, just make sure you are calling the `_async` functions instead of the regular ones if you are using a non-blocking driver!

Asyncmongo

The Asyncmongo backend is written around the asyncmongo driver developed by bit.ly and is closely designed around Pymongo.

Using the Asyncmongo back end is similar to any asynchronous task in Tornado development:

```
class thisHandler(MojoRequestHandler, MojoAuthMixin, SessionMixin_Async):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        this_user = yield gen.Task(Users.find_one_async,{'username':'martin'})

        if this_user:
            print 'Returned user: ', this_user.username

        self.render('template.html', this_user=this_user)
```

The back end exposes the base functions as listed above (`find`, `find_one`, `save`, `insert`, `delete`) and can then be accessed via the model objects as described in the `Auth.ObjectMapper.ModelPrototype`, which exposes the asyncmongo driver functions as <function_name>_async() instead of the direct association.

PyMongo

The pymongo backend is written around the driver written by the creators of MongoDB, usage is sraightforward enough:

```
class thisHandler(MojoRequestHandler, MojoAuthMixin, SessionMixin_Async):

        def get(self):

            this_user = Users.find_one({'username':'martin'})

            if this_user:
                print 'Returned user: ', this_user.username

            self.render('template.html', this_user=this_user)
```

The back end exposes the base functions as listed above (`find`, `find_one`, `save`, `insert`, `delete`) and can then be accessed via the model objects as described in the `Auth.ObjectMapper.ModelPrototype`, which exposes the pymongo driver functions.

2.3 Mojo.ObjectMapper

The Mojo object mapper is not strictly database-bound ORM, the purpose of the ORM is to offer data integrity and validation as part of the data solution. As Mojo is mainly built around MongoDB (for now), the models offer a way to ensure some consistency in a schemaless database.

The Object Mapper defines two main things: `Fields` that can be used to set up validation rules on data types and `Models`, collections of fields that will validate all of their containing fields and map to dictionaries under the hood.

`Models` come with multiple database functions (`save`, `find`, `find_one`, `delete`, `delete_bulk`) that also have asynchronous variants (`save_async`, `find_async`, `find_one_async`, `delete_async`, `delete_bulk_async`) that need to be called depending on the underlying database back-end that has been chosen.

2.3.1 Base Field Type

This is the base field type that all others inherit from, it can be subclassed to create custom model fields to extend your models.

class `Mojo.ObjectMapper.FieldPrototype.`**`Field`**(*value=None*, *allow_empty=True*, *default=None*, *friendly=''*, ***kwargs*)

`FieldPrototype.Field` is a parent class for the Fields exposed in `ObjectMapper.Fields`, Offering base functionality and validation options: #. validate_type - validates to ensure the base_type class variable is met #. validate_is_null - ensures that the property is not null (if set)

Usage: New Field types will inherit from this class:

```
from Mojo.ObjectMapper.FieldPrototype import Field

#Subclass Field to create a new field type
class StringField(Field):
    base_type = unicode #Must be set to the base type of object you want to save

    #Define your validation methods here and override the validate function to include them:

    def validate_max_length(self):
        #e.g. let's allow this field to have a maximum length
        if self.max_length != 0:
            if len(self.value) > self.max_length:
                raise ValueError('%s must be shorter than %i characters' % (self.__class__.__nam
                                                                            self.max_length))
            else:
                return True

    def validate(self):
        #Call the parent validation methods
        super(StringField, self).validate()

        #Run our own validation methods
        if self.max_length != 0:
            self.validate_max_length()
```

For a full example see the source for `Mojo.ObjectMapper.Fields.StringField` as the above example omits the `__init__()` overrides that establish required properties.

When defining fields it is also possible to inherit from an existing field type, for example, the `FloatField` field type inherits from the `IntegerField` type, this means we can chain functionality together for more finer-grained subtypes.

Methods to override

- `validate_type`: For example, if you have a subtype you would like to take advantage of or do explicit typecasting, see the source for `StringField` which coerces strings to unicode
- `validate`: To run your custom value validations
- `get_value`: To return the correct value back to the user

get_value()

Returns the value stored in the field (string representations will be shown if you print the model, override `__str__` and `__unicode__` to change this behaviour.

validate()

Will run all the validation rules, traditionally this will be overridden in a field class depending on what validation logic has been added to the Field

validate_is_null()

Validates if the value is empty. Will throw `ValueError` if `allow_empty` is false.

validate_type()

Basic type validation, this is not strict - it will validate subtypes as well, override this if you want to add coercion to your model (see the `StringField` and `FloatField` for an example)

2.3.2 Default Mojo Field Types

These fields are the default field types that come with Mojo and are used throughout the framework.

class `Mojo.ObjectMapper.Fields.`**`BooleanField`**(*value=None*, *allow_empty=True*, *default=None*, *friendly=''*, ***kwargs*)

Stores True or False data.

Validation methods

- *None*

`base_type`
alias of `bool`

`get_value`()
Returns the value stored in the field (string representations will be shown if you print the model, override `__str__` and `__unicode__` to change this behaviour.

class `Mojo.ObjectMapper.Fields.`**`DateTimeField`**(*value=None*, *allow_empty=True*, *default=None*, *friendly=''*, ***kwargs*)

DateTime object storage.

Validation methods

- *None*

`base_type`
alias of `datetime`

class `Mojo.ObjectMapper.Fields.`**`FloatField`**(**args*, ***kwargs*)

Stores Float values.

Validation methods

- Same as `IntegerField`

Note: Will coerce integers into float values.

`base_type`
alias of `float`

class `Mojo.ObjectMapper.Fields.`**`IntegerField`**(**args*, ***kwargs*)

Stores integer (`int`) data.

Validation methods

- `max_value`: maximum value of the integer
- `min_value`: minimum value of the stored value

`base_type`
alias of `int`

class `Mojo.ObjectMapper.Fields.`**`ListField`**(*of=None*, **args*, ***kwargs*)

Stores a list of objects, objects in the list can be of type 'Mojo.ObjectMapper.ModelPrototype.Model' and it will try to expand and retrieve the relevant value from each on access.

Validation methods

- *None*

`base_type`
alias of `list`

class `Mojo.ObjectMapper.Fields.`**`ObjectIDField`**(*value=None*, *allow_empty=True*, *default=None*, *friendly=''*, ***kwargs*)

ObjectId storage - required for storing data in MongoDB, base type is `bson.objectid.ObjectId`.

Validation methods

- *None*

`base_type`

alias of `ObjectId`

class `Mojo.ObjectMapper.Fields.`**`StringField`**(**args*, ***kwargs*)

Unicode string field for text based storage.

Validation methods

- `max_length`: Checks for the maximum length of the string, raises ValueError if not met.

`base_type`

alias of `unicode`

class `Mojo.ObjectMapper.ModelPrototype.`**`EmbeddedModelField`**(*to*, ***kwargs*)

EmbeddedField type for models that enables embedding of documents into the model representation.

Validation methods

- *None*

Note: This type expects a Model base type and is in a different namespace, to use `EmbeddedModelField` you need to import from the `ModelPrototype` namespace:

```
from ObjectMapper.ModelPrototype import EmbeddedModelField
```

`base_type`

alias of `Model`

2.3.3 Models

`Models` represent collections of `Fields`, and are subclasses of `dict`, so are easy to implement into Mongo-style queries which use BSON-style data structures.

class `Mojo.ObjectMapper.ModelPrototype.`**`Model`**(*data=None*)

Basic Model class that defines the behaviours of Model objects, the class enables simple definition:

```
class MyTestModel(Model):
    this_field =        StringField()
    another_field =     IntegerField()
    whatever_field =    BooleanField()
```

The class will evaluate `validate()` on each field when the value is retrieved and maps all values to an internal `dict` (as it subclasses `dict`) it behaves a little like one by enabling [x] and dot-style assignment and retrieval to values.

Models are recursive, so should produce a valid dict with `EmbeddedField` fields that are nested as they all expose the `get_value()` function.

Models ca be instantiated form a dictionary, so if you have som raw data you want to store in a model, that's fine:

```
class MyTestModel(Model):
    this_field =        StringField()
    another_field =     IntegerField()
    whatever_field =    BooleanField()

my_data = {
            'this_field':       "Hello World",
            'another_field':    42,
            'whatever_field':   True
          }

new_model_instance = MyTestModel(my_data)

print new_model_instance
#Output: {'this_field':"Hello World", 'another_field':42, 'whatever_field':True}
```

delete()
Removes an item form the database, takes a model as input (blocking).

delete_async(**args*, ***kwargs*)
Removes an item form the database, takes a model as input (non-blocking).

classmethod delete_bulk(*klass*, *to_delete*)
Delete items in a batch operation (blocking) - takes a list of models as input

classmethod delete_bulk_async(**args*, ***kwargs*)
Delete items in a batch operation (non-blocking) - takes a list of models as input

classmethod find(*klass*, **args*, ***kwargs*)
Find a list of objects in the database, takes standard PyMongo-style input parameters (blocking):

```
users = User.find({}) #gets all users in the database
print users
```

classmethod find_async(**args*, ***kwargs*)
Find a list of objects in the database, takes standard PyMongo-style input paramaters (non-blocking):

```
users = yield gen.Task(User.find_async,{}) #gets all users in the database
print users
```

classmethod find_one(*klass*, **args*, ***kwargs*)
Find a single object in the database, takes standard PyMongo-style input parameters (blocking):

```
thisUser = User.find_one({'username':username}) #Gets a specific user from the DB
```

classmethod find_one_async(**args*, ***kwargs*)
Find a single object in the database, takes standard PyMongo-style input parameters (non-blocking):

```
thisUser = yield gen.Task(User.find_one,{'username':username}) #Gets a specific user from th
```

get_value()
Returns the value of the model - this is a `dict` - if you're having trouble getting data out of the model, calling this function will access the dictionary directly.

classmethod insert(*klass*, *to_insert*)
Insert a model into the database (blocking).

Usage:

```
from models import MyTestModel

my_data = {
        'this_field':       "Hello World",
        'another_field':    42,
            'whatever_field':   True
        }

MyTestModel.insert(MyTestModel(my_data))
```

This will validate the fields before committing them to the database, without having to instantiate a new model instance.

classmethod insert_async(**args*, ***kwargs*)

Insert a model into the database (non-blocking).

Usage:

```
from models import MyTestModel

my_data = {
        'this_field':       "Hello World",
        'another_field':    42,
        'whatever_field':   True
        }

yield gen.Task(MyTestModel.insert_async, MyTestModel(my_data))
```

This will validate the fields before committing them to the database, without having to instantiate a new model instance.

save()

Saves the model instance to the database (blocking), unlike insert() this is an instance method so needs to be called by and instantiated model instance:

```
from models import MyTestModel

my_data = {
        'this_field':       "Hello World",
        'another_field':    42,
        'whatever_field':   True
        }

model_instance = MyTestModel(my_data)
model_instance.save()
```

save_async(**args*, ***kwargs*)

Saves the model instance to the database (non blocking), unlike insert() this is an instance method so needs to be called by and instantiated model instance:

```
from models import MyTestModel

my_data = {
        'this_field':       "Hello World",
        'another_field':    42,
        'whatever_field':   True
        }

model_instance = MyTestModel(my_data)
yield gen.Task(model_instance.save_async)
```

validate()

Validates the entire model, is called when _get_value() is called.

2.4 Mojo Form Helpers

Mojo offers convenience functions to make handling forms with the database easier. At it's core these functions are wrappers arounf basic wtForms functionality to make Form management easier.

class `Mojo.Forms.MojoFormHelper.`**`Form`**(*formdata=None*, *obj=None*, *prefix='‘*, *locale_code='en_US'*, ***kwargs*)

A wrapper around the standard `wtForms Form` object that integrates with tornado. Modified slightly for compatibility with Mojo models - you can pre-populate a form and populate a model using this Form class instead of the standard wtForms Form.

Example:

```
class SigninForm(Form):
    email = EmailField('email')
    password = PasswordField('password')

class SigninHandler(RequestHandler):
    def get(self):
        form = SigninForm(self.request.arguments, locale_code=self.locale.code)
```

classmethod **`populate_from_model`**(*klass*, *model*, *ignore=*[ ])

Populates a form from a model, if you want to ignore some fields, use the `ignore` list (of strings) to set which fields not to be iterated over.

`populate_model_from_data`(*modelObj*, *ignore=*[ ])

Will populate a model from the data provided by a form. Use the `ignore` list (must all be strings) to ignore data keys passed back by your form.

`Mojo.Forms.MojoFormHelper.`**`model_as_form`**(*model*, *initObj=None*, *ignore=*[ ], *override={}*)

Will take a model (non instantiated) object and produce a `WTForm` Form instance, this function only works one-way.

`model_as_form` can take three parameters:

- `initObj` - If you want to inittialise the form with data, pass it an instance of the model in question and it will do it's best to instantiate it
- `ignore` - If you want to exclude a list of fields from the automated conversion you can specify them as a list of strings corresponding to the name of the field in your model
- `override` - To specify a custom wtForm field type, pass these as a dictionary of `{"field name":FieldType,}`

***note*: The override function takes either classes or instantiated objects, so if you want to pass through custom validators** as part of the override, simply specify them in the override as if you would in a standard Form definition.

Usage:

```
from Mojo.Forms.MojoFormHelper import model_as_form

# Let's get some data from our imaginary blog app
posts = yield gen.Task(BlogPost.find_async, {'published':True}, sort=[('date_published',1)])
a_post = posts[0]

# Define a custom override set, the first is a base class,
# the second is an instance with my custom validators:

overrides = {'post_body': TextAreaField,
             'post_intro': TextAreaField('Your intro text', [validators.required(),
                                            validators.length(max=10)]}

# Create the form - we will ignore tags and
# comments (these are Lists, and not implemented),
# and override with our custom fields (StringFields are by
# default TextFields, but we want TextAreas):

thisForm = model_as_form(BlogPost, initObj=a_post, ignore=['tags', 'comments'], override=ove

# ...Now do something with the form
```

The system will automatically transform required model fields to required Form fields, no other validation checks are passed through yet.

For the 'friendly name' o your form field, Mojo will use the `friendly` Mojo Model Form parameter, if none is present, Mojo will use the name of the Modle Field.

2.5 Mojo.RequestHandlers

The `MojoRequestHandler` is the basic `RequestHandler` class that you use in your application instead of the standard Tornado `RequestHandler`:

```
class pageHandler(MojoRequestHandler, MojoAuthMixin, SessionMixin_Sync):
#You don't need to use the Mixins, but they make life a lot easier.

    def get(self):
        #Do some cool stuff

        self.render('login.html', error=None)
```

class `Mojo.RequestHandlers.MojoHandler.`**`MojoRequestHandler`**(*application*, *request*, ***kwargs*)

The `MojoRequestHandler` is the starting point of any request in Mojo, subclass this for every URL you want to serve in Mojo.

The `MojoRequestHandler` has some handy properties:

- `application`: This is a reference to the server object, and give access to project settings
- `application.mojo_settings`: All variables declared in your `settings.py` file
- `db`: the database Session object (for direct access to the DB you can get to the driver through `db._db.<function>`)

If you are using the `MojoAuthMixin` or `SessionMixin`, the request handlers expand with many more management functions to make life easier for the developer.

2.6 Mojo.SocketHandlers

Mojo has added some simple convenience to TornadIO2 SocketConnections by adding two key dictionaries to the framework. These dictionaries are populated (and depopulated) as users connect to the site.

The dictionaries that are made available are:

- CURRENT_SESSIONS: Keyed by the session ID of the user
- LOGGED_IN_SESSIONS: Keyed by the user ID of the (logged in) user (requires Mojo.Auth to be enabled).

Usage:

```
from Mojo.SocketHandlers.BaseSocketHandler import MojoSocketHandler, CURRENT_SESSIONS, LOGGED_IN_SESS

#Setup your socket connections here (remember to subclass MojoSocketHandler)
class SocketConnectionHandler(MojoSocketHandler):
    def on_message(self, msg):
        pass
```

class `Mojo.SocketHandlers.BaseSocketHandler.`**`MojoSocketHandler`**(*session*, *endpoint=None*)

The `MojoSocketHandler` overrides the `on_open` and `on_close` events of the standard `TornadIO2 SocketConnection` to expose some simple session data (`self.session_id` and `self.logged_in`).

`MojoSocketHandler` will store connections in a global dictionary called `CURRENT_SESSIONS`, all connections are keyed by the user `session_id` (generated automatically on request).

You can import `CURRENT_SESSIONS` to broadcast to all users, or emit messages to specific users based on their `session_id`.

If you are using the `Mojo.Auth` module, and the cookie `logged_in` is set (i.e. the user is logged in), then another global dictionary: `LOGGED_IN_SESSIONS` is populated with the connection, keyed by the `user_id` of the user. This makes common tasks much simpler to manage such as:

- Broadcasting to all connected clients
- Broadcasting to only logged in users
- Online 'status' checking of logged in users - e.g. for chat.
- Event-based messaging, e.g. with signals

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

m

Mojo.Auth.AuthManager, 13
Mojo.Auth.Helpers, 14
Mojo.Auth.Mixins.MojoAuthMixin, 17
Mojo.Auth.models, 15
Mojo.Auth.SessionManager, 15
Mojo.Backends.base_interface, 21
Mojo.Forms.MojoFormHelper, 28
Mojo.ObjectMapper.FieldPrototype, 23
Mojo.ObjectMapper.Fields, 24
Mojo.RequestHandlers.MojoHandler, 30
Mojo.SocketHandlers.BaseSocketHandler, 31