1. Tutorial

This tutorial introduces **MongoEngine** by means of example — we will walk through how to create a simple **Tumblelog** application. A tumblelog is a blog that supports mixed media content, including text, images, links, video, audio, etc. For simplicity's sake, we'll stick to text, image, and link entries. As the purpose of this tutorial is to introduce MongoEngine, we'll focus on the data-modelling side of the application, leaving out a user interface.

1.1. Getting started

Before we start, make sure that a copy of MongoDB is running in an accessible location — running it locally will be easier, but if that is not an option then it may be run on a remote server. If you haven't installed MongoEngine, simply use pip to install it like so:

```
$ python -m pip install mongoengine
```

Before we can start using MongoEngine, we need to tell it how to connect to our instance of **mongod**. For this we use the `connect()` function. If running locally, the only argument we need to provide is the name of the MongoDB database to use:

```
from mongoengine import *

connect('tumblelog')
```

There are lots of options for connecting to MongoDB, for more information about them see the Connecting to MongoDB guide.

1.2. Defining our documents

MongoDB is *schemaless*, which means that no schema is enforced by the database — we may add and remove fields however we want and MongoDB won't complain. This makes life a lot easier in many regards, especially when there is a change to the data model. However, defining schemas for our documents can help to iron out bugs involving incorrect types or missing fields, and also allow us to define utility methods on our documents in the same way that traditional ORMs do.

In our Tumblelog application we need to store several different types of information. We will need to have a collection of **users**, so that we may link posts to an individual. We also need to store our different types of **posts** (eg: text, image and link) in the database. To aid navigation of our Tumblelog, posts may have **tags** associated with them, so that the list of posts shown to the user may be limited to posts that have been assigned a specific tag. Finally, it would be nice if **comments** could be added to posts. We'll start with **users**, as the other document models are slightly more involved.

1.2.1. Users

Just as if we were using a relational database with an ORM, we need to define which fields a `User` may have, and what types of data they might store:

```
class User(Document):
    email = StringField(required=True)
    first_name = StringField(max_length=50)
    last_name = StringField(max_length=50)
```

This looks similar to how the structure of a table would be defined in a regular ORM. The key difference is that this schema will never be passed on to MongoDB — this will only be enforced at the application level, making future changes easy to manage. Also, the User documents will be stored in a MongoDB *collection* rather than a table.

1.2.2. Posts, Comments and Tags

Now we'll think about how to store the rest of the information. If we were using a relational database, we would most likely have a table of **posts**, a table of **comments** and a table of **tags**. To associate the comments with individual posts, we would put a column in the comments table that contained a foreign key to the posts table. We'd also need a link table to provide the many-to-many relationship between posts and tags. Then we'd need to address the problem of storing the specialised post-types (text, image and link). There are several ways we can achieve this, but each of them have their problems — none of them stand out as particularly intuitive solutions.

1.2.2.1. Posts

Happily MongoDB *isn't* a relational database, so we're not going to do it that way. As it turns out, we can use MongoDB's schemaless nature to provide us with a much nicer solution. We will store all of the posts in *one collection* and each post type will only store the fields it needs. If we later want to add video posts, we don't have to modify the collection at all, we just *start using* the new fields we need to support video posts. This fits with the Object-Oriented principle of *inheritance* nicely. We can think of `Post` as a base class, and `TextPost`,

`ImagePost` and `LinkPost` as subclasses of `Post`. In fact, MongoEngine supports this kind of modeling out of the box — all you need do is turn on inheritance by setting `allow_inheritance` to True in the `meta`:

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)

    meta = {'allow_inheritance': True}

class TextPost(Post):
    content = StringField()

class ImagePost(Post):
    image_path = StringField()

class LinkPost(Post):
    link_url = StringField()
```

We are storing a reference to the author of the posts using a `ReferenceField` object. These are similar to foreign key fields in traditional ORMs, and are automatically translated into references when they are saved, and dereferenced when they are loaded.

1.2.2.2. Tags

Now that we have our Post models figured out, how will we attach tags to them? MongoDB allows us to store lists of items natively, so rather than having a link table, we can just store a list of tags in each post. So, for both efficiency and simplicity's sake, we'll store the tags as strings directly within the post, rather than storing references to tags in a separate collection. Especially as tags are generally very short (often even shorter than a document's id), this denormalization won't impact the size of the database very strongly. Let's take a look at the code of our modified `Post` class:

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
```

The `ListField` object that is used to define a Post's tags takes a field object as its first argument — this means that you can have lists of any type of field (including lists).

Note

We don't need to modify the specialized post types as they all inherit from `Post`.

1.2.2.3. Comments

A comment is typically associated with *one* post. In a relational database, to display a post with its comments, we would have to retrieve the post from the database and then query the database again for the comments associated with the post. This works, but there is no real reason to be storing the comments separately from their associated posts, other than to work around the relational model. Using MongoDB we can store the comments as a list of *embedded documents* directly on a post document. An embedded document should be treated no differently than a regular document; it just doesn't have its own collection in the database. Using MongoEngine, we can define the structure of embedded documents, along with utility methods, in exactly the same way we do with regular documents:

```
class Comment(EmbeddedDocument):
    content = StringField()
    name = StringField(max_length=120)
```

We can then store a list of comment documents in our post document:

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

1.2.2.4. Handling deletions of references

The `ReferenceField` object takes a keyword *reverse_delete_rule* for handling deletion rules if the reference is deleted. To delete all the posts if a user is deleted set the rule:

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User, reverse_delete_rule=CASCADE)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

See `ReferenceField` for more information.

> **Note**
>
> MapFields and DictFields currently don't support automatic handling of deleted references

1.3. Adding data to our Tumblelog

Now that we've defined how our documents will be structured, let's start adding some documents to the database. Firstly, we'll need to create a `User` object:

```
ross = User(email='ross@example.com', first_name='Ross', last_name='Lawley').save()
```

Note

We could have also defined our user using attribute syntax:

```
ross = User(email='ross@example.com')
ross.first_name = 'Ross'
ross.last_name = 'Lawley'
ross.save()
```

Assign another user to a variable called `john`, just like we did above with `ross`.

Now that we've got our users in the database, let's add a couple of posts:

```
post1 = TextPost(title='Fun with MongoEngine', author=john)
post1.content = 'Took a look at MongoEngine today, looks pretty cool.'
post1.tags = ['mongodb', 'mongoengine']
post1.save()

post2 = LinkPost(title='MongoEngine Documentation', author=ross)
post2.link_url = 'http://docs.mongoengine.com/'
post2.tags = ['mongoengine']
post2.save()
```

Note

If you change a field on an object that has already been saved and then call `save()` again, the document will be updated.

1.4. Accessing our data

So now we've got a couple of posts in our database, how do we display them? Each document class (i.e. any class that inherits either directly or indirectly from `Document`) has an `objects` attribute, which is used to access the documents in the database collection associated with that class. So let's see how we can get our posts' titles:

```
for post in Post.objects:
    print(post.title)
```

1.4.1. Retrieving type-specific information

This will print the titles of our posts, one on each line. But what if we want to access the type-specific data (link_url, content, etc.)? One way is simply to use the `objects` attribute of a subclass of `Post`:

```
for post in TextPost.objects:
    print(post.content)
```

Using TextPost's `objects` attribute only returns documents that were created using `TextPost`. Actually, there is a more general rule here: the `objects` attribute of any subclass of `Document` only looks for documents that were created using that subclass or one of its subclasses.

So how would we display all of our posts, showing only the information that corresponds to each post's specific type? There is a better way than just using each of the subclasses individually. When we used `Post`'s `objects` attribute earlier, the objects being returned weren't actually instances of `Post` — they were instances of the subclass of `Post` that matches the post's type. Let's look at how this works in practice:

```
for post in Post.objects:
    print(post.title)
    print('=' * len(post.title))

    if isinstance(post, TextPost):
        print(post.content)

    if isinstance(post, LinkPost):
        print('Link: {}'.format(post.link_url))
```

This would print the title of each post, followed by the content if it was a text post, and "Link: <url>" if it was a link post.

1.4.2. Searching our posts by tag

The `objects` attribute of a `Document` is actually a `QuerySet` object. This lazily queries the database only when you need the data. It may also be filtered to narrow down your query. Let's adjust our query so that only posts with the tag "mongodb" are returned:

```
for post in Post.objects(tags='mongodb'):
    print(post.title)
```

There are also methods available on `QuerySet` objects that allow different results to be returned, for example, calling `first()` on the `objects` attribute will return a single document, the first matched by the query you provide. Aggregation functions may also be used on `QuerySet` objects:

```
num_posts = Post.objects(tags='mongodb').count()
print('Found {} posts with tag "mongodb"'.format(num_posts))
```

1.4.3. Learning more about MongoEngine

If you got this far you've made a great start, so well done! The next step on your MongoEngine journey is the full user guide, where you can learn in-depth about how to use MongoEngine and MongoDB.