Embedding or Linking

There are no joins in MongoDB but you do get powerful and flexible dictionary style documents over simple rows. When it comes to modelling data in MongoDB you have a choice, you can embed that data and keep it local or you can link to that data but there is a cost of an extra join but it can be easier to query

Embedded Documents

MongoEngine has an `EmbeddedDocument` class for describing the schema of embedded data. Essentially its very similar to a `Document` but there is less `meta` magic handling indexes and setting up `QuerySet Managers`.

Comments

So lets add `Comments` as an embedded list to our `Post` model. First define a comment:

In [1]:
```
import datetime
import mongoengine as db

conn = db.connect('tumblelog')
conn.drop_database('tumblelog')

class Comment(db.EmbeddedDocument):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=
    body = db.StringField(verbose_name="Comment", required=True)
    author = db.StringField(verbose_name="Name", max_length=255, required=

    def __unicode__(self):
        return (u"comment by %s" % self.author) if self.author else "New C

Comment()
```

Out[1]:
```
<Comment: New Comment>
```

Now update the `Post` document definition to add in comments. As this is a one-to-many relationship - a post can have 0 or many documents we use the `ListField` type and pass in the `EmbeddedDocumentField` for the individual comments:

In [2]:
```
class Post(db.Document):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    body = db.StringField(required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))

    def __unicode__(self):
        return unicode(self.title) or u"New Post"
```

Lets ensure this works, add some data and see how we can query against the embedded data.

In [3]:
```
Post(title="mongoengine post",
     slug="mongoengine-post",
     body="Welcome to Europython 2012!").save()

comment_1 = Comment(author="Ross",
```

```
                        body="Nice post thanks")
comment_2 = Comment(author="Bob",
                        body="Florence rocks!")

Post.objects.update(push_all__comments=[comment_1, comment_2])

post = Post.objects.first()
post.comments
```

Out[3]: [<Comment: comment by Ross>, <Comment: comment by Bob>]

Querying against embedded data

Now we have some comments, we can look at querying comments to find documents that match. To query an `EmbeddedDocument` we use *parent_field_name__embedded_field_name*=expected_value:

In [4]:
```
Post.objects(comments__author="Ross")
```

Out[4]: [<Post: mongoengine post>]

As you can see that matches all documents where Ross has authored a comment.

Updating Embedded data

This is similar to how you updated an ordinary document eg:
update_operation__*parent_field_name__embedded_field_name*=value.

When dealing with lists you can also match against position by using the `positional operator $`. As symbols aren't allowed in keyword arguments (without using dictionaries) you can use **S** as the positional match. The positional operator will only update the first list element that matches the query. eg:

In [5]:
```
Post.objects(comments__author="Ross").update(set__comments__S__author="Roz
Post.objects.first().comments
```

Out[5]: [<Comment: comment by Rozza>, <Comment: comment by Bob>]

Paginating comments

MongoDB's query language is focused on returning documents. However, you can reduce how much data is sent over the wire by paginate through a list fields by using the `slice` operator to skip and limit the returned list items:

In [6]:
```
post = Post.objects.fields(slice__comments=[1,1]).first()
post.comments
```

Out[6]: [<Comment: comment by Bob>]

Relations

MongoEngine also handles relationships similar conceptially similar to Foreign Keys, however, you cannot query through a foreign key. In the background MongoEngine will collect all the references up and then issue extra queries to get the data.

Author

Lets add a `User` class and add `Author` to the Post.

In [7]:
```
class User(db.Document):
    email = db.StringField(required=True)
    first_name = db.StringField(max_length=50)
    last_name = db.StringField(max_length=50)

ross = User(email="ross@10gen.com", first_name="Ross", last_name="Lawley")
ross
```

Out[7]:
```
<User: User object>
```

In [8]:
```
class Post(db.Document):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    body = db.StringField(required=True)
    author = db.ReferenceField('User', required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))

    def __unicode__(self):
        return unicode(self.title) or u"New Post"

# Migrate!
Post.objects.update(set__author=ross)
```

Exercises

I. Create a couple of posts in your database and add comments to each post.
II. Write a loop that paginates through the comments of a post.
III. Write some code to find all comments by a particular author.
IV. What are the advantages and disadvantages of Embedding data?
V. What happens if you try to query through a ReferenceField?