



U N I V E R S I D A D
COMPLUTENSE
M A D R I D

MongoEngine

Gestión de la Información en la Web
Enrique Martín - emartinm@ucm.es
Grados de la Fac. Informática

Motivación

- MongoDB carece de **esquema estricto**, por lo que una colección puede almacenar datos muy dispares.
- Esta flexibilidad **facilita** el **desarrollo** y permite acomodar de manera sencilla **cambios** en los datos almacenados.
- Sin embargo, tener campos con nombres y tipos ligeramente diferentes pero puede llevar al **caos**.
- Por tanto es importante definir y seguir un **esquema implícito**, aunque éste tenga cierto grado de libertad.

MongoEngine

- MongoEngine es una **capa intermedia** entre el programa Python y MongoDB para conseguir:
 - Seguir disfrutando del esquema flexible de MongoDB.
 - Garantizar un mínimo de coherencia.
- Para ello, MongoEngine realiza una **transformación objeto-documento** (Object-Document Mapping, ODM) entre objetos Python y los documentos de las colecciones MongoDB.

MongoEngine

- Con MongoEngine definimos **clases** Python junto con su **esquema**: campos esperados y tipos de datos.
- MongoEngine **traduce** los objetos a documentos MongoDB **garantizando** que cumplen el esquema definido.
- MongoDB **almacena** los documentos como siempre, ignorando completamente el esquema definido en MongoEngine.
- MongoEngine proporciona métodos para realizar consultas a Mongo y recuperar directamente los objetos Python de manera sencilla.

Definición del esquema

connect()

- Antes de realizar cualquier tarea con MongoEngine es necesario conectar con el servidor **mongod**.
- Para ello se usa la función **connect()**:

```
from mongoengine import connect  
connect('nombre_bd')
```
- El parámetro es el nombre de la base de datos donde se crearán las colecciones necesarias.
- Por defecto se conecta a **localhost** en el puerto **27017**.

connect()

- MongoEngine admite más parámetros a la hora de conectarse al servidor MongoDB:

```
connect (
    name='db',
    username='user',
    password='12345',
    host='92.168.1.35'
)
```

Definición de documentos

- Para definir el ODM declararemos **clases Python** que **heredan** de clases de MongoEngine (Document, DynamicDocument o EmbeddedDocument)
- Dentro de cada clase declararemos los **campos** que existen, su **tipo** y otra **información** (si es clave primaria, es obligatorio, etc.)

Definición de documentos

- Definir una clase **Asignatura** con 3 campos:
 - **nombre**: cadena de texto, obligatorio
 - **código**: entero, clave primaria → obligatorio
 - **temario**: lista de cadenas de texto, no obligatorio

```
class Asignatura(Document) :  
    nombre = StringField(required=True)  
    codigo = IntField(primary_key=True)  
    temario = ListField(StringField())
```

Document

- Las clases que heredan de **Document** almacenan los objetos de ese tipo en una **colección** con el mismo nombre que la clase (en minúsculas).
- Si se añaden **campos adicionales** no declarados a un objeto, éstos **no se almacenarán** en la base de datos.
- Por lo tanto, **Document** es interesante para objetos cuya composición se conoce perfectamente y no se esperan cambios en el futuro.

Document

- Definición:

```
class Asignatura(Document) :  
    nombre = StringField(required=True)  
    codigo = IntField(primary_key=True)  
    temario = ListField(StringField())
```

- Creación y almacenamiento:

```
a = Asignatura(nombre="GIW",  
               codigo=803348,  
               temario=["XML y JSON", ...])  
a.profesor = "Enrique" # Campo adicional  
a.save()
```

- El documento almacenado en la colección **asignatura** no contendrá el campo **profesor**, ya que no aparecía en el esquema de **Asignatura**:

```
{  
    "_id" : 803348,  
    "nombre" : "GIW",  
    "temario" : [ "XML y JSON", ...]  
}
```

DynamicDocument

- Las clases que heredan de **DynamicDocument** almacenan los objetos de ese tipo en una **colección** con el mismo nombre que la clase (en minúsculas), al igual que Document.
- Como diferencia, los **campos adicionales** añadidos a un objeto sí **se almacenarán** aunque no hayan sido definidos en el esquema.
- **DynamicDocument** es interesante en objetos para los que conocemos su composición básica pero que son susceptibles de ser **ampliados** en el futuro.

DynamicDocument

- Definición:

```
class Asignatura(DynamicDocument):  
    nombre = StringField(required=True)  
    codigo = IntField(primary_key=True)  
    temario = ListField(StringField())
```

- Creación y almacenamiento:

```
a = Asignatura(nombre="GIW",  
               codigo=803348,  
               temario=["XML y JSON", ...])  
a.profesor = "Enrique" # Campo adicional  
a.save()
```

- El documento almacenado en la colección **asignatura** sí contendrá el campo **profesor**, aunque no aparecía en el esquema de **Asignatura**:

```
{  
    "_id" : 803348,  
    "nombre" : "GIW",  
    "temario" : [ "XML y JSON", ...],  
    "profesor" : "Enrique"  
}
```

Campos

Campos

- Por defecto los **campos** almacenados en MongoDB tienen el **mismo nombre** que los atributos definidos en MongoEngine.
- Este comportamiento se puede cambiar con el parámetro **db_field** en cada campo:

```
class Asignatura(Document):  
    nombre = StringField(db_field='name')  
    ...
```

Campos

- Los campos también aceptan otros parámetros:
 - **required**: el campo es obligatorio (por defecto False).
 - **default**: valor por defecto que tomará el campo si no se asigna. Puede ser un *callable* (p. ej. una función sin argumentos) que calcula el valor por defecto.
 - **unique**: el campo es único (por defecto False).
 - **unique_with**: para definir combinaciones de campos como valores únicos (por defecto None).
 - **primary_key**: el campo es la clave primaria **_id**.

Campos

- También se puede limitar a que un campo contenga **valores** de un **listado fijo**.
- Para ello se usa el parámetro **choices**, que acepta una lista de valores legítimos:

```
class Persona(Document):  
    sexo = StringField(choices=['H', 'M'])  
    ...
```

- Si el campo toma un valor no especificado en **choices**, la validación fallará (lanzará una excepción **ValidationError**) y el objeto no se almacenará en MongoDB.

Tipos de campos

- MongoEngine permite utilizar **muchos** tipos de campos diferentes. Veremos solo unos pocos (más información en las Referencias):
 - BooleanField
 - IntField, FloatField
 - StringField
 - ComplexDateTimeField
 - ListField
 - EmailField, URLField

BooleanField

- Define un campo que solo puede contener un valor booleano: True o False.

```
class Persona(Document):  
    parado = BooleanField  
    ...
```

- Tened **cuidado** porque Python puede evaluar casi cualquier expresión a un booleano, por lo habrá validaciones que sorprendentemente tendrán éxito:

```
p = Persona(parado="SI", ...)
```

Establecerá **parado** a **True** porque **bool("SI") → True**

IntField, LongField y FloatField

- **IntField** y **LongField** definen campos para números **enteros** de **32 y 64 bits** respectivamente.
- **FloatField** define un campo para números en **coma flotante**.
- Los 3 campos permiten acotar los valores:
 - **min_value**: valor mínimo
 - **max_value**: valor máximo

IntegerField, LongField y FloatField

- Almacenar la **edad** de una Persona en años, su **peso** en kilogramos y el **número de pasos** que ha dado en toda su vida:

```
class Persona(Document):  
    edad = IntegerField(min_value=0,  
                        max_value = 200)  
    peso = FloatField(min_value=0)  
    pasos = LongField(min_value=0)  
    ...
```

StringField

- Define un campo que contiene una **cadena de texto**.
- Acepta varios parámetros para especificar los valores válidos:
 - **min_length**: longitud mínima.
 - **max_length**: longitud máxima.
 - **regex**: expresión regular que debe cumplir su contenido.

StringField

- Para almacenar el NIF de una persona, añadiríamos un campo tipo cadena:

```
Class Persona(Document):  
    nif = StringField(  
        max_length = 9,  
        regex = "[0-9]+[A-Z]")  
    )  
    ...
```

ComplexDateTimeField

- Define un campo para contener una **fecha** con **precisión** de microsegundo:
`'YYYY,MM,DD,HH,MM,SS,NNNNNN'`
 - Ej: '1900,01,01,15,38,52,138464'
- En MongoDB se almacenan como **cadenas** de texto.
- Estos campos se pueden **comparar** con `>`, `<`, `>=`, etc. ya que son cadenas de texto, y el orden **lexicográfico** funciona adecuadamente.

ListField

- Define una **lista** de valores, todos del tipo de datos especificado y cumpliendo las restricciones impuestas.
- Lista de booleanos:
ListField(BooleanField)
- Lista de cadenas de al menos 3 caracteres:
ListField(StringField(min_length=3))
- Lista de enteros entre 0 y 100:
ListField(IntField(min_value=0, max_value=100))

EmailField y URLField

- **EmailField** permite definir campos de texto que deben contener un e-mail bien formado.
- **URLField** define campos de texto con un URL bien formado. Adicionalmente puede verificar que el recurso existe con el parámetro **verify_exists**.

```
class Persona(Document):  
    email = EmailField  
    web = URLField(verify_exists = True)  
    ...
```

Anidar y referenciar

Anidar y referenciar

- En MongoDB hay dos técnicas para **relacionar** documentos:
 - **Anidar** uno dentro de otro.
 - **Referenciar** uno desde otro usando su **_id**.
- MongoEngine nos va a permitir definir este tipo de relaciones usando campos de tipo:
 - **EmbeddedDocumentField**
 - **ReferenceField**

Anidar

- Para anidar una clase como campo interno otra, la clase anidada debe heredar de **EmbeddedDocument** o **DynamicEmbeddedDocument**.
- Estas clases **no generarán una colección** dedicada para almacenar objetos, ya que estarán anidadas dentro de otras.
- Los **campos adicionales** no declarados en una clase se **almacenarán** en MongoDB si se hereda de **DynamicEmbeddedDocument**.
- Si se hereda de **EmbeddedDocument** los campos adicionales se **ignorarán** al volcar a MongoDB.

Anidar

- Para definir un campo con un objeto anidado, usaremos el tipo **EmbeddedDocumentField**:

```
class Direccion(EmbeddedDocument) :  
    calle    = StringField(required=True)  
    numero  = IntField(min_value=0)
```

```
class Persona(Document) :  
    dir=EmbeddedDocumentField(Direccion)  
    ...
```

Referenciar

- Al referenciar, incluimos el **identificador** de un documento de una **colección externa** en uno de los campos del documento.
- Para incluir referencias en MongoEngine usaremos campos **ReferenceField**:

```
class Mascota (Document) :  
    nombre = StringField
```

```
class Persona (Document) :  
    mascota = ReferenceField(Mascota) #Otra clase  
    jefe = ReferenceField("self") #Misma clase
```

Referenciar

- Cuando un objeto tiene un campo referenciado, debemos elegir **qué hacer** cuando el objeto referenciado se **elimina** de la colección externa.
- Para ello usamos el parámetro **reverse_delete_rule** del campo **ReferenceField**:
 - **DO_NOTHING (0)**: no hacer nada (por defecto).
 - **NULLIFY (1)**: elimina el campo que contiene la referencia.
 - **CASCADE (2)**: elimina los documentos que contienen la referencia.
 - **DENY (3)**: impide borrar documentos de la colección externa si están referenciados.
 - **PULL (4)**: si el campo donde está la referencia es un ListField, elimina dicha referencia de la lista.

Manipulación de objetos

Esquema de ejemplo

- Consideremos un esquema simple con 3 tipos de objetos:
 - **Direccion** está **anidado** dentro de **Persona**.
 - **Mascota** está **referenciado** desde **Persona**.

```
class Direccion(EmbeddedDocument) :  
    calle = StringField(required=True)  
    numero = IntField(min_value=0)
```

```
class Mascota(Document) :  
    nombre = StringField(required=True)
```

```
class Persona(Document) :  
    nombre = StringField(required=True)  
    dir = EmbeddedDocumentField(Direccion)  
    email = EmailField  
    mascota = ReferenceField(Mascota)
```

Crear objetos

- Para **crear objetos** usaremos la **sintaxis usual** de Python, pasando como **parámetros** de constructor los **campos** definidos en el esquema.

```
mascota1 = Mascota(nombre="Fifi")
mascota2 = Mascota(nombre="Koki")
direccion = Direccion(calle="Mayor", numero=8)
persona = Persona(nombre="Eva", dir=direccion,
                  email="eva@mail.com", mascota=mascota1)
```

- Si los parámetros siguen el mismo orden que la definición de los campos, se puede omitir el nombre del campo:

```
persona = Persona("Eva", direccion,
                  "eva@mail.com", mascota1)
```

Insertar objetos

- **Crear** objetos Python **no inserta** documentos en **MongoDB**.
- Para insertar el documento en MongoDB es necesario invocar al método **save()** sobre el objeto.

```
mascota1.save()  
mascota2.save()  
persona.save()
```

- Invocar `save()` sobre objetos anidados (como **Direccion**) no realiza ninguna escritura en la base de datos.

```
direccion.save() #No tiene efecto
```

Insertar objetos

- Los objetos **referenciados** deben existir en MongoDB **antes** de invocar a `save()`.

```
# mascota1.save()  
# Olvidamos escribir 'mascota1'  
persona.save()
```

ValidationError:

ValidationError (Persona:None)

**(You can only reference documents
once they have been saved to the
database: ['mascota'])**

Actualizar objetos

- El método `save()` **actualizará** un objeto si éste ya existe en MongoDB.
- Se entenderá que un objeto **existe** si su **clave primaria** aparece en la colección.

```
m1.save()  
p.save() #Inserta el documento  
p.email = "eva@eva.com"  
p.save() #Actualiza el documento
```

Validación

- MongoEngine realiza la **validación de los campos** cuando se invoca a **save()**, **no al crear el objeto**.

```
m = Mascota() # No ocurre nada  
m.save() # Lanza excepción
```

ValidationError:

ValidationError (Mascota:None)

(Field is required: ['nombre'])

Eliminar objetos

- Para borrar un objeto se invoca a su método **delete()**.
- **delete()** **no tiene** ningún **efecto** si el objeto no ha sido **insertado previamente**.
- También se puede **eliminar** la **colección completa** donde se almacena un objeto mediante el método **drop_collection()**.

Consultas

Recorrer colecciones

- Para consultar una colección, usaremos el **atributo objects de su clase**.
- El atributo **objects** es un objeto de tipo **QuerySet** que nos permite **iterar sobre los objetos**:

```
m1 = Mascota("Fifi")  
m2 = Mascota("Koki")  
m1.save()  
m2.save()
```

```
for e in Mascota.objects:  
    print(e.nombre)  # 'e' es un objeto Mascota
```

- La salida producida será:

```
Fifi  
Koki
```

Igualdad sobre campos

- El atributo **objects** admite **parámetros** para definir consultas más precisas.
- En el caso más sencillo son **igualdades sobre campos**:
 - # Mascotas con nombre "Fifi"
`Mascota.objects(nombre="Fifi")`
 - # Personas con 10 años
`Persona.objects(edad=10)`
- El resultado sigue siendo un **QuerySet iterable**.

Consultas sobre campos

- MongoEngine admite **operadores sobre los campos** para afinar las consultas, al igual que MongoDB.
- Estos operadores se **concatenan** al nombre del campo con **doble subrayado __**:
 - `Mascota.objects(nombre__ne="Fifi")` → distinto
 - `Persona.objects(edad__gt=10)` → mayor que
 - `Persona.objects(edad__lte=10)` → menor o igual a
 - `Persona.objects(nombre__in=["Eva", "Pepe"])` → campo toma valores en un listado
- Existen más operadores, ver más información en **Referencias**.

Consultas sobre campos

- Para referirse a **campos de objetos anidados** se usa el **doble subrayado** `__` en lugar del punto:

```
Persona.objects(dir__calle="Mayor")  
#Personas que viven en la calle Mayor
```

- Los campos de objetos anidados pueden ser complementados con **operadores de consulta**:

```
Persona.objects(dir__numero__gt=6)  
#Personas que viven en edificios con  
#numeracion mayor a 6
```

Conjunción y disyunción

- Para establecer varias **condiciones** que se deben cumplir **a la vez** solo es necesario pasar varios parámetros al atributo **objects**

- # Personas que se llaman Eva y viven en
la calle Mayor

```
Persona.objects(nombre='Eva',  
                 dir__calle='Mayor')
```

- # Personas que viven en la calle Mayor y
tienen más de 25 años

```
Persona.objects(dir__calle='Mayor',  
                 edad__gt=25)
```

Conjunción y disyunción

- Para representar condiciones **disyuntivas** es necesario usar **objetos Q** y combinarlos con el **operador |**.

- Los objetos Q contienen condiciones de consulta:

```
# Personas llamadas Pep o con 5 años
Persona.objects( Q(edad=5) |
                  Q(nombre='Pep') )
```

- Los objetos Q también se pueden combinar mediante **conjunción con &**:

```
# Personas que o bien se llaman Pep o bien
# tienen 5 años y además viven en la calle Mayor
Persona.objects( Q(nombre='Pep') |
                  (Q(edad=5) & Q(dir__calle='Mayor')) )
```

Consultas `__raw__`

- Siempre es deseable representar las consultas siguiendo la propia sintaxis de MongoEngine.
- En casos excepcionales es posible pasar como parámetro una **consulta pymongo** directamente mediante el parámetro `__raw__`:

```
Persona.objects(  
    __raw__={ 'edad' : { '$gt' : 5 } }  
)
```


Proyectar campos

- Para reducir el número de campos devueltos por la consulta utilizaremos el método **only()**:
 - # Todas las personas, solo **nombre**
`Persona.objects.only('nombre')`
 - # Personas mayores de 5 años, solo **nombre** y **dirección**
`Persona.objects(edad__gt=5).only('nombre', 'dir')`
- Al usar `only()`, los **campos omitidos** de los objetos devueltos contendrán **None**.

Limitar resultados

- Se puede **limitar** de manera muy sencilla los **resultados** obtenidos utilizando los *slices* de Python:
 - 5 primeros objetos Persona con nombre 'Eva':
`Persona.objects(nombre='Eva')[:5]`
 - Personas de la posición 10 a la 19:
`Persona.objects[10:20]`
 - Primera persona de 55 años:
`Persona.objects(edad=55)[0]`

Limitar resultados

- Además de la sintaxis de *slices*, para limitar resultados se pueden utilizar métodos de QuerySet:
 - 5 primeros objetos Persona con nombre 'Eva':
`Persona.objects(nombre='Eva').limit(5)`
 - Personas de la posición 10 a la 19:
`Persona.objects.skip(10).limit(10)`
 - Primera persona de 55 años:
`Persona.objects(edad=55).first()`

Limitar resultados

- Si el resultado de una consulta es **exactamente un objeto** (p.ej. buscar un usuario existente por DNI), se puede usar **get()**:
 - `m = Mascota.objects.get(nombre='Fifi')`
- Si el resultado no es exactamente un único objeto, **get() lanzará excepciones**:
 - `DoesNotExist`
 - `MultipleObjectsReturned`

Ordenar resultados

- Los resultados obtenidos en una consulta se pueden **ordenar** con el método **order_by()**:
 - Todas las **mascotas** ordenadas por **nombre ascendente**:
`Mascota.objects.order_by('nombre')`
 - Todas las **mascotas** ordenadas por **edad descendente** y luego por **nombre descendente**:
`Mascota.objects.order_by('-edad', '-nombre')`
 - **Gatos** ordenados por **edad ascendente**:
`Mascota.objects(tipo='Gato').order_by('edad')`

Contar el número de resultados

- Para contar el **número de resultados** en el **QuerySet** generado por una consulta se pueden usar dos técnicas:
`ps = Persona.objects(nombre='Eva')`
 - `len(ps)`
 - `ps.count()`
- La opción **len()** es la **preferida**, al usar la sintaxis usual de Python.

Aspectos avanzados

Métodos en clases

- Se pueden **añadir métodos** en las clases que definen el ODM de MongoEngine, para facilitar su utilización en el programa:

```
class Mascota(Document):
    nombre = StringField(required=True)
    edad = IntField(min_value=0, required=True)
    tipo = StringField(choices=["Gato", "Perro"],
                       required=True)

    def es_gato_joven(self) :
        return (self.tipo == "Gato") and
               (self.edad < 5)
```

- Podemos invocar estos métodos en los objetos creados y en los recuperados por MongoEngine:

```
for e in Mascota.objects:
    if e.es_gato_joven() :
        (...)
```


Validación personalizada

- Los campos predeterminados de MongoEngine ya proporcionan **validación por defecto**:
 - Tipo de dato almacenado
 - Longitud/valores válidos
 - Expresiones regulares
- Sin embargo en ocasiones necesitaremos **validaciones personalizadas**. P.ej. *los nombres de gato tienen solo 4 letras*.
- En estas situaciones implementaremos el método **clean()**, que se invocará automáticamente al llamar a **save()** y antes de realizar la inserción/actualización.
- El método **clean()** lanzará la excepción **ValidationError** si los datos no son consistentes.

Validación personalizada

```
class Mascota(Document):  
    nombre = StringField(required=True)  
    edad = IntField(min_value=0, required=True)  
    tipo = StringField(choices=["Gato", "Perro"],  
                        required=True)
```

```
def clean(self):  
    if (self.tipo == "Gato") and  
        (len(self.nombre) != 4):  
        raise ValidationError("Los nombres de  
gato deben tener 4 letras")
```

- Además de validación, el método `clean()` también puede realizar **limpieza en los datos**.

Herencia de clases

- Las clases que hemos definido heredaban directamente de **Document** y similares.
- Si en nuestras clases tenemos relaciones de herencia (p.ej. Usuario > UsuarioAdministrador) podemos expresarlas en MongoEngine:
 - Todos los documentos se almacenarán juntos en la **colección de la clase padre**.
 - Podremos realizar **consultas** sobre **subclases concretas**.

Herencia de clases

```
class Ave(Document) :  
    nombre = StringField(primary_key=True)  
    meta    = {'allow_inheritance': True}
```

```
class Aguila(Ave) :  
    alt = IntField(min_value=0)
```

```
class AguilaReal(Aguila) :  
    vel = IntField(min_value=0)
```

- Es imprescindible **activar** la **herencia** en el campo **meta** de la clase principal.

Herencia de clases

```
a1 = Ave(nombre='Piticli')  
a2 = Aguila(nombre='Veloz', alt=5)  
a3 = AguilaReal(nombre='Rayo', alt=4, vel=150)  
a1.save()  
a2.save()  
a3.save()
```

- Este código **inserta 3 documentos** en la **colección ave**:

```
{ "_id": "Piticli", "_cls": "Ave" }
```

```
{ "_id": "Veloz", "_cls": "Ave.Aguila", "alt": 5 }
```

```
{ "_id": "Rayo", "_cls": "Ave.Aguila.AguilaReal",  
  "alt": 4, "vel": 150 }
```

Herencia de clases

- A la hora de realizar consultas, podemos usar las distintas clases como antes:
 - Todas las águilas reales:
`AguilaReal.objects`
 - Águilas (y águilas reales) de al menos 3 metros:
`Aguila.objects (alt__gte=3)`
 - Todas las aves con nombre acabado en 'i':
`Ave.objects (nombre__endswith='i')`

Ejercicio con MongoEngine

Pedidos en una tienda

- Consideremos una tienda en la que almacenamos información de los **usuarios** y de los **pedidos realizados por cada uno**.
 - De los **usuarios** almacenamos su nombre, NIF, teléfono y una lista de pedidos realizados.
 - Un **pedido** está formado por un número de pedido, una fecha, un total y una lista de ítems.
 - Cada **ítem** contiene el nombre de producto, el número de unidades, el precio por unidad y el precio total.

Pedidos en una tienda

- Diseñar el **esquema MongoDB** para almacenar esta información sabiendo que:
 - Los **ítems** están **anidados** en la lista dentro del pedido.
 - Los **pedidos** están **anidados** en la lista dentro del usuario.
 - Todos los **campos** son **obligatorios**, y queremos establecer el **valor mínimo** y **expresión regular** (básica) donde sea aplicable.
 - También queremos tener **validación personalizada** para asegurar que los **totales** en los **ítems** y los **pedidos** están bien calculados.

Pedidos en una tienda

- Escribe el **código Python** necesario para **insertar dos usuarios**:
 - El usuario 'pepe' ha realizado 2 pedidos, cada uno con 2 ítems.
 - La usuaria 'eva' ha realizado 1 pedido de 3 ítems.

Referencias

Referencias

- Tutorial básico sobre MongoEngine:
<http://docs.mongoengine.org/tutorial.html>
- Definición de esquemas:
<http://docs.mongoengine.org/guide/defining-documents.html>
- Manejo de objetos:
<http://docs.mongoengine.org/guide/document-instances.html>
- Consultas con MongoEngine:
<http://docs.mongoengine.org/guide/querying.html>

Referencias

- QuerySet:
<http://docs.mongoengine.org/apireference.html#mongoengine.queryset.QuerySet>
- Operadores de consulta:
<http://docs.mongoengine.org/guide/querying.html#query-operators>