UNIVERSIDAD COMPLUTENSE MADRID

Persistencia en Python: ODM

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

Motivación

- Durante la ejecución de un programa, los datos están almacenados en distintos **objetos** que creamos y modificamos.
- Para conseguir persistencia entre ejecuciones, necesitamos almacenar los datos de esos objetos usando algún componente que garantice su conservación.
- La solución más adecuada es utilizar una **BD** para conseguir esta persistencia.

Motivación

- Volcar y recuperar un objeto a una base de datos suele involucrar un cambio en la representación que debemos manejar de manera manual:
  - BD relacional (Oracle, MySQL, PostgreSQL...): un objeto puede involucrar varias tablas diferentes.
  - BD documentos (MongoDB, OrientDB...): un objeto puede ser almacenado en más de una colección.
- Este problema se conoce como **adaptación de impedancias** (*impedance mismatch*)

Motivación

- Una solución interesante es usar una capa intermedia para realizar esta traducción **objeto ←→ BD** de manera automática.
- Dependiendo del modelo de datos de la BD subyacente:
  - **ORM**: *Object-Relational mapping (sqlalchemy)*
  - **ODM**: *Object-Document mapping (mongoengine)*
- Una vez definida esta traducción, el programador puede olvidarse de la BD y trabajar únicamente con los objetos.
- Abstraer la BD permite cambiar de SGBD de manera transparente.

MongoEngine

- MongoEngine proporciona persistencia de objetos Python basándose en MongoDB, una BD NoSQL muy popular y con grandes capacidades de escalado.
- En MongoEngine definimos las **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.
- MongoEngine proporciona métodos para recuperar directamente los objetos Python de manera sencilla.

Definición del esquema

Instalar e iniciar MongoDB

- **El primer paso para usar MongoEngine** es disponer de un servidor **MongoDB ejecutándose y escuchando conexiones**.
- La versión gratuita de MongoDB se llama "*Community Server*" y se puede descargar de https://www.mongodb.com/try/download/community.
- Si MongoDB se ha instalado con el instalador, tendréis una entrada en el menú de inicio para iniciar la BD. También es posible que se lance automáticamente como servicio cada vez que se arranca el sistema.
- Si habéis descargado la versión comprimida, deberéis lanzar manualmente el servidor ejecutando el binario **mongod** (o **mongod.exe** en Windows)

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`
- Dentro de cada clase declararemos los **campos** que existen, su **tipo** y otra **información** (si es clave primaria, es obligatorio, etc.)
- Los campos serán **atributos de la clase** a los que asignaremos ***objetos campo*** **de MongoEngine**

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 MongoDB.
- Si se añaden **campos adicionales** a un objeto que no han sido declarados en su esquema, 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()
```

- Al almacenar esta asignatura no se almacenará el campo **profesor**, ya que no aparecía en el esquema de **Asignatura.**

DynamicDocument

- Las clases que heredan de **DynamicDocument** permiten almacenar los objetos en MongoDB.
- A diferencia de **Document**, 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()
```

- La asignatura almacenada **sí conserva** el campo **profesor**, aunque no aparecía en el esquema de **Asignatura.**

Campos

Campos

- Los clases de ***campos*** aceptan varios parámetros en su constructor:
  - **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.

Campos

- También se puede restringir 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):
      pref = StringField(choices=['A','B','C'])
      …
  ```
- Si el campo toma un valor no especificado en **choices**, la validación fallará (lanzará una excepción **ValidationError**) y el objeto no se almacenará.
  ```

Clave

- En MongoEngine todos los objetos ***almacenados*** tendrán una clave primaria que sirve para realizar búsquedas.
- Si se ha definido un campo como **primary_key**, ese campo será la clave.
- Si no se ha definido ninguna clave, se generará una al almacenar el objeto de manera *automática*.
- En cualquiera de los dos casos, la clave primaria será accesible a través del atributo **pk** del objeto. Ej:

```
a = Asignatura(…)
print(a.pk)       # Ambos atributos
print(a.codigo)   # son alias
```

Tipos de campos

- MongoEngine proporciona **muchas** clases 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(required=True)
  …
```

- Tened **cuidado** porque Python puede evaluar casi cualquier expresión a un booleano, por lo que 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

IntField, 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 = IntField(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, que es una cadena de longitud 9 formada por 8 dígitos y una letra mayúscula, añadiríamos un campo tipo cadena:

```
class Persona(Document):
  nif = StringField(
          min_length = 9,
          max_length = 9,
          regex = "[0-9]{8}[A-Z]")
  )
  …
```

StringField y regex

- **Importante**: la expresión regular usada en ***regex*** se debe cumplir **en algún lugar de la cadena**, no obligatoriamente en la cadena completa.
- P. ej., cadenas como “1ola”, “Ga3z” o “123a” encajarían con el campo
  `StringField(min_length=4, max_length=4,regex="[0-9]+")`
  porque: 1) tienen longitud mínima 4
  2) tienen longitud máxima 4
  3) alguna subcadena encaja con `[0-9]+`
- Para que la expresión regular **cubra toda la cadena** debemos forzar la longitud del encaje (entre llaves) o usar anclas para indicar inicio (^) y final ($).
- Posibles definiciones de cadenas de **exactamente 4 dígitos**:
  - `StringField(min_length=4, max_length=4,regex="[0-9]{4}")`
  - `StringField(min_length=4, max_length=4,regex="^[0-9]+$")`
  - `StringField(regex="^[0-9]{4}$")`

ComplexDateTimeField

- Define un campo para contener una **fecha** con **precisión** de microsegundos:
  `'YYYY,MM,DD,HH,MM,SS,NNNNNN'`
  - Ej: '1900,01,01,15,38,52,138464'
- Estos campos se pueden **comparar** con >, <, >=, etc.

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 una URL bien formada. 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

- MongoEngine permite dos técnicas para **relacionar** unos objetos con otros:
  - **Anidar** uno dentro de otro, de tal manera que el objeto interno no puede existir por sí solo sino dentro del objeto externo → campo **EmbeddedDocumentField**
  - **Referenciar** uno desde otro usando su identificador, pudiendo existir ambos de manera independiente → campo **ReferenceField**

Anidar

- Para anidar una clase como campo interno de otra, la clase anidada debe heredar de **EmbeddedDocument** o **DynamicEmbeddedDocument**.
- Los **campos adicionales** no declarados en una clase se **almacenarán** si se hereda de **DynamicEmbeddedDocument.**
- Si se hereda de **EmbeddedDocument** los campos adicionales se **ignorarán** al almacenar la información.

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 una referencia (usando el identificador) a otro objeto.
- 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 se elimina el objeto referenciado.
- Para ello usamos el parámetro **reverse_delete_rule** del campo **ReferenceField:**
  - **DO_NOTHING (0)**: no hacer nada (por defecto).
  - **NULLIFY (1)**: actualiza el valor a *null*.
  - **CASCADE (2)**: elimina todos los objetos que contienen esa referencia.
  - **DENY (3)**: impide borrar objetos si están referenciados por algún otro.
  - **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** del constructor los **campos** definidos en el esquema usando **expresamente su nombre**.

```
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)
```

Insertar objetos

- **Crear** objetos en Python **no los almacena automáticamente** en MongoDB.
- Para almacenar un objeto 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, ya que no pueden existir salvo dentro de otros objetos:

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

Insertar objetos

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

```
    # mascota1.save()
    # Olvidamos salvar '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 ha sido almacenado previamente.
- 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

- **Importante:** 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 almacenado **previamente**.
- También se pueden **eliminar** todos los objetos de una clase mediante el método **drop_collection():**

```
Persona.drop_collection()
```

- Solo se pueden eliminar objetos que existen por sí mismos, es decir, que no son anidados.

Consultas

Recuperar objetos

- Para recuperar objetos de una clase, usaremos el **atributo *objects* de dicha 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.
- 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 (__)**:

```
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
# numeración 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** (*query*) 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 llamadan 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')) )
```

Limitar resultados

- Se pueden **limitar** 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

- 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 en caso de empate por **nombre ascendente**:

    ```
    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 usa **len()**:

```
qs = Persona.objects(nombre='Eva')
print(len(qs))
```

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():
    print('Little meow')
```

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() debe lanzar 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(f"Los nombres de gato deben
              tener 4 letras: {self.nombre}")
```

- Además de validación, el método clean() también está pensado para realizar **limpieza en los datos** (p.ej. corregir el código de país de “ESP” a “ES”).

Validación personalizada

- **Importante:** cuando existe el método `clean()` éste se invoca **antes de realizar las validaciones estándar** de los campos.
- Por lo tanto, el código de `clean()` se puede ejecutar sobre **campos mal formados** porque todavía no se han validado.
- Ejemplo sin `clean()`:

```
class Persona(Document):
  dni = StringField(required=True)

p = Persona(dni=43.5)
p.save()
# ValidationError (Persona:None) (StringField only
# accepts string values: ['dni'])
# ***VALIDACIÓN ESTÁNDAR***
```

Validación personalizada

- Ejemplo con `clean()`:

```
class Persona(Document):
  dni = StringField(required=True,
                    min_lenght=9, max_length=9)

  def clean(self):
    letra = self.dni[8]
    if (letra not in string.ascii_uppercase):
      raise ValidationError('Falta la letra del DNI')

p = Persona(dni=43.5)
p.save()
# TypeError: 'float' object is not subscriptable
# Error en ‘self.dni[8]’
```

Validación personalizada

- En ambos casos el almacenamiento del documento en MongoDB se aborta debido a una excepción, pero...
- ...es necesario que los errores de validación generen excepciones **ValidationError,** que es la manera de informar de esta situación en MongoEngine.
- Una solución sencilla es forzar una validación manual invocando a `self.validate(clean=False)` nada más comenzar el método `clean()`. Si algún campo no cumple los requisitos se lanzará una excepción **ValidationError**.
- Sin embargo, esto hace que las validaciones estándar se ejecuten dos veces: la primera de manera manual vez dentro de `clean()` y la segunda vez de manera automática tras la ejecución de `clean()`.

Validación personalizada

- Es imprescindible el parámetro `clean=False` porque por defecto `validate()` invocaría a `clean()`, lo que generaría un bucle de llamadas recursivas.

```
def clean(self):
  self.validate(clean=False)
  letra = self.dni[8]
  # Ahora está garantizado que self.dni es una
  # cadena de texto de 9 caracteres
  if (letra not in string.ascii_uppercase):
    raise ValidationError('Falta la letra del DNI')
```

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