Ohhm

Metaprogramación

Object Mapper en Ruby

Para ejercitar metaprogramación, vamos a desarrollar un (prototipo de) mapeador de objetos a una persistencia. Especificamente para guardar nuestros objetos en una base de datos orientada a documentos, llamada MongoDB.

MongoDB tiene una librería para Ruby con su documentación acá
Sin embargo la API de esa librería es más bien básica.
Lo que nosotros queremos es desarrollar un “mini-framework” que permita definir la persisencia de nuestros objetos de una forma “declarativa” y que el frameworks se ocupe de traducir eso a llamados a Mongo.

Nota sobre Mongo

No se asusten si no cursaron “Estrategias de Persistencia”, mongo tiene features avanzados, pero no son necesarios para nuestro TP.
Por el contrario es una base bastante simple comparada con las relacionales.
Si ya conocen mongo pueden saltearse esto, si no lo conocen puende ver el Anexos **“Intro a Mongo”**

Puntos del TP

Vamos a hacer un mini-framework que nos permita declarar en nuestras clases de dominio información y nos provee funcionalidad para poder guardar nuestros objetos en mongo, sin tener que escribir código que le pegue directo a la librería de mongo.

Para eso vamos a ir agregándole funcionalidad en forma iterativa.

Declaración de Fields y Collection

Primeramente vamos a hacer algo un poco pavo por ahora. Como Ruby no provee una sintaxis para declarar variables de instancia, vamos a introducir una propia nuestra que permita modelar la idea de que un objeto tendrá fields.
Más adelante le vamos a agregar más información o funcionalidad a los “fields”.
Por ahora se deberá poder escribir esto

```
class Pregunta
    include MongoDocument

    field :autor, String
    field :contenido, String
end
```

Obviamente que “field” es un método que no existe en Ruby y uds lo tienen que proveer.
MongoDocument tampoco existe. Es un module nuevo.

Luego tengo que poder

a) **Tener accessors automáticos**

```
 =          .new
.autor = "Eduardo"
```

```
.contenido = "Cual es el radio de la tierra"
```

b) Reflejar los fields por metaprogramación

Dada una clase poder preguntarle sus fields y que me de la lista (no hace falta soportar herencia!)

```
expect(        .fields).to eq ([:autor, :contenido])
```

c) Nombre de Collection

Cada clase se va a persistir en una colección distinta en Mongo.
Por **defecto** el framework deberá usar el nombre de la clase, convirtiendo a minúscula la primera letra, y agregando una “s” al final. Esto debe ser accesible desde el metamodelo también

```
expect(        .collectionName).to eq (“preguntas”)
expect(    .collectionName).to eq (“casas”)
```

Pero se deberá poder customizar desde la clase

```
class Pregunta
  include MongoDocument

  collection :preguntinhas

  field :autor, String
  field :contenido, String
end
```

Luego

```
expect(        .collectionName).to eq (“preguntinhas”)
```

Ojo al comparar symbols con strings. A veces van a necesitar traducir de un symbol a un string con blah.to_s

Métodos de persistencia (instancias y clase)

Queremos que cada uno de nuestros “modelos” de dominio tenga métodos “piola” para manejar la persistencia. Por ejemplo para

- **objeto.save:** guardar el objeto
- **object.update:** actualizarlo
- **objeto.remove:** borrarlo de la colección de mongo

Vamos de a poco.

a) asHash

Tenemos que poder guardar un objeto. Pero para eso, conviene primero implementar la idea de que cada uno de estos objetos sabe traducirse a un hash.

```
 =          .new
.autor = "Eduardo"
.contenido = "ABC?"

expect( .asHash).to eq ({"autor"    "Eduardo", "contenido"
"ABC?"})
```

Traducir a hash significa ***<u>utilizar la información de los "fields" para generar un diccionario con esos fields y sus valores.</u>***
No significa recorrer todas las variables de instancia.
O sea, necesitamos reflection para que dado los nombres de los fields obtener el valor actual en el objeto.

Para acotar el TP no vamos a soportar objetos anidados. Es decir, por ejemplo, que una Pregunta pueda tener un field "autor" de tipo "Usuario" y ese a su vez tenga sus propios fields. Simplemente nuestras clases de dominio van a necesitar ser objetos simples con tipos básicos.

Nota: Ojo con las conversiones entre String Symbol. Y las convenciones de que las instVars se llaman con @. Hay que jugar un poco para, dado el nombre del field obtener el valor de la variable de instancia

b) save()

Ahora sí, tenemos que poder guardar un objeto así

```
 =          .new
.autor = "Eduardo"
.contenido = "ABC?"
```

```
.save()
```

El save() eventualmente utiliza la API de mongo, Ejemplo

```
client.colleccion.insert_one(objeto.asHash)
```

Ver el “Find” en el Anexo para aprender a usar Mongo desde consola y ver cómo está la base.

Ver Anexo ***“Manejando la Conexión a Mongo”*** para entender cómo conectarse a mongo y dónde guardar el estado de esa conexión.

Sería interesante dividir el comportamiento en métodos chiquitos. Por ejemplo a cualquier objeto le puedo pedir la collecion de mongo

```
p = Pregunta.new
expect(p.collection).not_to eq (nil)
```

O incluso a la clase

```
expect(Pregunta.collection).not_to eq (nil)
```

Nota: Recordar que ya implementamos la idea de inferir el nombre de la colección en base al nombre de la clase !

b) Count

Antes de seguir avanzando con otros métodos necesitamos implementar el count, para empezar a tener tests de verdad con asserts.
Este a diferencia del save() no necesita de una instancia (todos los de “consulta” o búsqueda). Tenemos que poder hacer esto, es decir pedirselo a la clase

```
old = Pregunta.count()
Pregunta.new().save()
expect(Pregunta.count()).to eq (old + 1)
```

c) Generando Ids

Para seguir implementando funcionalidad de persistencia vamos a necesitar cambiar un poco el save().
Cada documento en mongo tiene un “id” único, que es un field llamado “_id” (guión bajo, id) en Mongo.
Si uno guarda un documento sin este valor, entonces mongo lo genera automáticamente.
Sin embargo hoy en día esto no funcionaría, porque si bien en Mongo se le genera un id, a nuestro objeto referenciado por “p” no se le actualiza el _id con ese valor.

```
  =          .new
 .save()
expect( ._id).not_to be nil
```

Entonces necesitamos que nuestro framework se haga cargo de generar ese “id” en el momento de guardar el objeto por primera vez (save()). Y actualice la instancia (el objeto referenciado por “p” en este caso) con ese valor. Así nos evitamos tener que ir a buscarlo a mongo luego de un save()

La forma de generar ese id en ruby es

```
    =     ::         .new
```

Ahora el test anterior debería funcionar.

d) remove

Ahora sí, podemos implementar una versión básica del remove, como método de instancia de los modelos

```
 =          .new()
.save()
   =          .     ()

.remove()   # yeah!

   expect(          .     ()).to eq (    - 1)
```

Para eliminar un objeto en particular vamos a usar el método “delete_one” de mongo y matchear por el “_id” del objeto

```
    collection.delete_one({"_id"    @_id})
```

Otra forma de hacer un assert con más precisión sería

```
  =          .new()
 .save()
expect( .collection.    ({"_id"     ._id}).to_a).not_to be
be_empty

    =          .     ()

 .remove()

expect(         .     ()).to eq (    - 1)
expect( .collection.    ({"_id"     ._id}).to_a).to be_empty
```

Finders - Búsquedas

Vamos a implementar funcionalidad para poder hacer búsquedas. Vamos a necesitar dos funcionalidades

- **find():** el método “básico” para poder hacer cualquier búsqueda
- **findBy???():** capacidad de buscar por fields específicos, como si fueran muchos métodos sobrecargados.

Find simple

Como justamente queremos buscar instancias, va a tener que ser un método de clase. Queremos poder hacer algo así, es decir buscar “todo” (sin ningún filtro / query)

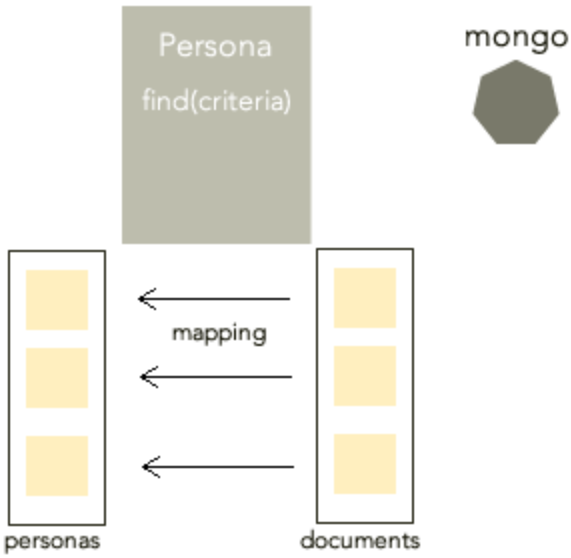
```
  =          .new
 .save()

        =          .    ()    // ACA

expect(       ).not_to be_empty
expect(       .any? {|        |         ._id ==  ._id}).to be
true
```

Pero también debería dejarnos buscar por un criterio, que va a ser un Hash “field” => valor
Ejemplo

```
Pregunta.find({"autor" => "Pepin"})
Pregunta.find({"autor" => "Pepin", "contenido" => "lalala"})
```

Nótese que el método find() debe entonces recibir un hash opcional, que va a ser el “criterio” de búsqueda, y deberá retornar una colección con los resultados.
De qué tipo van a ser los resultados ? del tipo *Pregunta* en este caso.
O sea, que el find() no puede devolver los objetos tal cual vienen de Mongo, porque esos serían objetos de bajo nivel, Documentos !

Y acá entonces es donde nuestro mapper realmente mapea.
Necesita implementar lógica para que dado un documento se cree una instancia de la clase actual (“*Pregunta”* por ej) y se le populen las variables de instancia.
Lógicamente esto requiere reflection



Nótese que lo que se mapea son los fields declarados en la clase Persona/Pregunta. Pero además hay que mapear el “_id” siempre.

Por otro lado, acá de nuevo no hay que volverse locos tratando de mapear objetos complejos ó anidados.
En un framework real habría que tener lógicas para convertir los valores que vienen mongo al tipo especificado en los “fields”. Nosotros simplemente vamos a asignar al valor como viene.

Finds dinámicos

El find que implementamos es el más básico, y “poderoso”, pero es medio molesto para búsquedas sencillas.
Entonces, varios frameworks implementan la idea de tener métodos “finds” dinámicos.

La idea es que haya muchos métodos “findBy” **virtuales**, es decir que tienen que existir para el que los llama, pero no necesariamente tienen que ser métodos reales. Y no queremos que los codifique cada usuario para cada una de sus clases.
Ejemplo

Dada nuestra clase Pregunta

```
class

 collection :preguntas

        :autor,
        :contenido,
        :tema,
end
```

Automáticamente uno debería poder hacer sin nada más, esto:

- `.findByAutor("Pepin")`
- `.findByContenido("Hola?")`
- `.findByTema("matematicas")`

Es decir buscar por cualquier de sus fields declarados.

Pero también poder buscar por más de un field !

- `.findBy`**`Autor`**`And`**`Tema`**`("Pepin", "matematicas")`
- `.findBy`**`Tema`**`And`**`Autor`**`("matematicas", "Pepin")`
- `.findBy`**`Autor`**`And`**`Tema`**`And`**`Contenido`**`("Pepin", "matematicas", "Hola?")`

BONUS:

Permitir una variante a esto para buscar un sólo objeto (el primero que matchee, o bien que tire error si se encontraron más de uno, pero si se encontró uno que retorne ese)
Simplemente por convención de nombre

- `.findOneByAutorAndTema("Pepin", "matematicas")`

Hooks

Al definir la persistencia de nuestros objetos de dominio es común tener ciertas validaciones o acciones a ejecutar antes del save() por ejemplo, o luego del save().
El framework entonces deberá proveer un mecanismo para esto, basado en convenciones

Before & After save()

La clase de dominio puede definir métodos siguiendo una convención de nombres

La convención es

- before_save()
- after_save()

Ambos métodos no deben recibir parámetros, ni devolver nada. Pueden lanzar una exception en para evitar el save, por ejemplo en el “before”

Ejemplo

```
class

 collection :preguntas

       :autor,
       :contenido,

 def            ()
                              , "Autor es requerido" if autor
== nil
 end
end
```

on_populate

Este hook sirve para ejecutar una lógica en cada objeto que el framework se trae de mongo. Por ejemplo cuando hacemos un find() esto trae los documentos de mongo y los convierte en instancias del dominio (como una Pregunta, Persona, etc). Bueno, para cada objeto, una vez que se le popularon las variables de instancia, queremos poder ejecutar un pedazo de código.

Para cambiar un poco y usar otros features de metaprogramación vamos a implementar esto de una manera alternativa a los hooks anteriores. En lugar de usar una convención de nombre, el framework debe proveer un método para agregarle un bloque

```
class


 on_populate {
   @populate_called = true
 }
 def
   @populate_called
 end
end
```

Ojo, nótese que el “on_populate” es un método que llamanos pasándole como parámetro un bloque.
Ese bloque en este ejemplo setea una variable de instancia. Dónde ?? si en realidad es un bloque ? Bueno, en la instancia sobre la que se está ejecutando.
Es decir que el bloque se va a tener que aplicar a cada una de las instancias (les suena ?)

Validaciones

Por último queremos tener validaciones.
Para simplificar el TP vamos a tener dos validaciones nada más

Validación de Requerido

Por otro lado debe ser posible especificar que un “field” es requerido.

Ejemplo, dada esta clase de negocio

```
class


        :autor,        , :required    true
        :contenido,
end
```

Notar que el “:required => true” es un mapa/hash/diccionario que estamos enviando como tercer parámetro al método “field”. En este caso tiene un único elemento “required => true”. Pero si uno quisiera podría modelar más características de los fields en un futuro como el valor por defecto, u otras validaciones como “min”, “max”, etc.

Queremos que automáticamente cuando alguien llama al setter se ejecute la validación y falle

```
   =          .new
 expect {  .autor = nil } .to raise_error("This field cannot be nil")
```

Ojo que esto significa que ahora ustedes necesitan meter código a los setters !
Si estaban usando “attr_accessors” ya no pueden seguir usándolo y deberán generar los setters uds. O bien encontrar una forma de que el setter los llame.

Validación de Tipo

Utilizando la información del tipo del field que ya tenemos, tendremos que además, validar que el objeto que se está seteando cumpla con ese tipo (o sea una subclase, o clase que tiene ese mixin).

```
class

       :autor,         , :required    true
end

class

end
```

Y un caso de error

```
   =          .new
 expect {  .autor = “Pepe” } .to raise_error("The field autor requires a object of type Persona, got a String instead")
```

Bonus

-
-
-

Resumen de Bonus

- Find “one” de los finds dinámicos.
- Conversiones:
    - Diseñar un esquema configurable de “conversiones” que en base al tipo del field declarado convierta los valores que vienen de mongo a un objeto Ruby. Por ejemplo para clases propias como Direccion, o Email.
- Validaciones:
    - Implementar validaciones de “min” y “max” para fields numéricos.
    - Permitir configurar si las validaciones se hacen en los “setters” o bien recién cuando se hace save() / update()
- Mapeo
    - Soportar objetos complejos anidados. Es decir que una clase como Pregunta tenga un field de un tipo también que se mapea a Mongo. Ejemplo Pregunta tenga un “autor” de tipo Usuario, que también se mapea a mongo y debe traerse.

Anexo

Intro a MongoDB

En lugar de “tablas” una base de datos de mongo tienen “colecciones”. Y cada colección tiene “documentos” (en lugar de registros). Los documentos son de tipo JSON.

Por si no conocen JSON es un formato para describir objetos en texto (aunque más que “objetos” son estructuras de datos, porque no se permiten tener métodos, sólo atributos). Ejemplo de JSON

```
{
    “nombre” : “Juan”,
    “apellido” : “Perez”,
    “edad” : 27,
    “vivo” : true
    “documento” :   {
            “numero” : 31.234.123
            “tipo”: “DNI”
     }
     “padres” :  [
          { “nombre” : “Jose” },
          { “nombre” : “Maria” }
     ]
```

```
}
```

Como se ve acá un objeto es como un Diccionario, o Mapa. Tiene claves y valores.
Esos valores pueden ser

- Strings
- Numeros
- Booleanos
- Otros objetos, por ej: el “documento”. Se pueden tener N niveles de anidamiento.
- Listas / Arrays. po ej: “padres” Y claro esos arrays pueden tener como elementos, números, strings, booleanos, objetos, u otros arrays. En nuestro caso tenemos dos objetos que sólo tienen “nombre”.

Eso es básicamente un JSON. No puede haber variables, ni métodos, ni código en sí. Sólo datos estructurados.

Para instalar mongo y el driver de ruby pueden ver el “Anexo” al final de este documento.

La documentación de la librería

Básicamente uno debe crear una conexión a una base de datos, y a partir de ahí tiene objetos que representan a las colecciones, y métodos para agregar un elemento (insert_one), para modificar (update), buscar (findOne) etc.

La mayoría de esos métodos reciben Diccionarios (Mapas) de ruby. Entonces justamente nuestro framework se deberá encargar de traducirlos a hashes.

Mongo

Para usar mongo necesitamos dos cosas

1. Instalar mongo en nuestras máquinas
   https://www.mongodb.com/download-center?jmp=nav#community
2. Instalar el driver de mongo para ruby en nuestro proyecto.

Para 2)

Editar Gemfile y agregar

```
        'https://rubygems.org'

gem 'rspec'
gem 'mongo'
```

Luego ejecutar “Tools > Bundler > Install”

Find en mongo

Una forma de interactuar con la base para ver su contenido es desde línea de comando

```
#conectamos a la base “preguntas”
mongo preguntas

# hacemos un find en la colección de “preguntas”
> db.preguntas.find({})
{ "_id" : ObjectId("5807be3bfbbb3d609ba27a26"), "autor" :
"Eduardo", "contenido" : "ABC?" }
```

Manejando la Conexión a Mongo

En varios métodos como save(), updates(), etc, vamos a necesitar tener una conexión a mongo. No queremos andar creando una conexión por cada ejecución.
Eso se puede resolver de varias maneras, y en un proyecto real uno querría tener la conexión configurable, de modo de conectarse a diferentes bases de datos, según si son los tests o la aplicación en producción, etc.
Nosotros no le vamos a dar bola a eso y vamos a hacer lo más sensillo posible.

Entonces una forma es tener la conexión (el objeto “client”) en algún lugar accesible globalmente desde los métodos tipo save(), remove(), etc.
Por ejemplo utilizando las variables de clase @@.
Acá va un ejemplo

```
module MongoDocument
 @@client = nil   # ruby te fuerza a declararla

 def self.client()
  if (@@client == nil)
    @@client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'preguntas')
  end
  @@client
 end
end
```

Luego desde cualquier lado para obtener una referencia podemos hacer:

```
MongoDocument.client
```