Chapter 1

A Tutorial about Voyage

This chapter describes a step by step tutorial showing the possibilities offered by Voyage an object to document mapper. We will use a simple but not trivial domain: super heroes, super powers and their equipments. You will learn how to save and retrieve objects.

1.1 What is Voyage

The working context of Voyage is the one of NoSQL databases. Relational databases work using a tabular metaphor (everything is a table) and they use Venn math (select, join, etc.). NoSQL do not.There are several kinds of NoSQL databases: object-oriented, document-oriented, column-oriented, or key-value storages. Each of them provides different advantages. Voyage focuses on document-oriented databases such as MongoDB or CouchDB.

Voyage is an object to document mapper for MongoDB. Voyage may be extended in the future to work with different back-ends such as CouchDB.

Here are the design guidelines that drove Voyage development.

- **It should be simple.** Voyage minimizes the descriptions to be given by the developer.
- **It should ensure object identity.** Voyage ensures that you cannot have inconsistencies by having one object reloaded with a different identity than the one it got.
- **It should provide error-handling.**
- **It should minimize communication.** Voyage implements a connection pool.

Voyage does not define a Voyage Query Language but use the underlying back-end query language. You have to use the MongoDB query language even if you can use blocks to define queries you can also use JSON dictionaries to express queries since MongoDB internally uses JSON.

1.2 Getting started

First you should load Voyage using its configuration as follows:

Loading Voyage

```
Metacello new
   smalltalkhubUser: 'Pharo' project: 'MetaRepoForPharo40';
   configuration: 'VoyageMongo';
   load.
```

Creating a connection

One you installed MongoBD, we can start to connect to the database as follows:

```
| repository |
repository := VOMongoRepository
      host: 'localhost'
      database: 'superHeroes'.
repository enableSingleton.
```

If you are not connected to a database, you can always use *in memory*repository (useful for prototyping your application).

```
| repository |
repository := VOMemoryRepository new.
repository enableSingleton
```

With this approach you can work as if you would be connected to a real database and later during your development you will be able to transparently switch mode.

Usually we define one single method to set up the repository. For example, we can add a class method to the class Hero that we will define just after.

```
Hero class >> setUpConnection
   | repository |
```

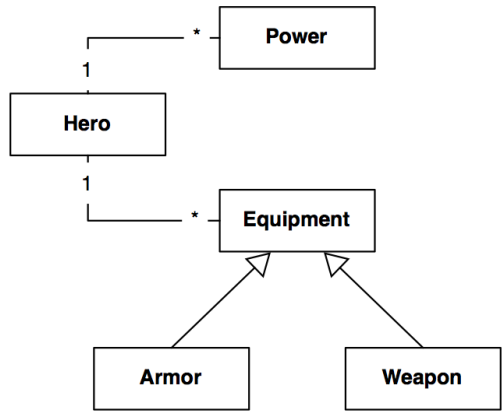

Figure 1.1: The model: SuperHeroes, SuperPowers and their Equipments.

```
repository := VOMongoRepository
    host: 'localhost'
    database: 'superHeroes'.
repository enableSingleton.
```

1.3 SuperHeroes

Now we can define a first version of our domain. Figure 1.1 shows the model that we will use for this tutorial.

Heroes

Let us define the class Hero.

```
Object subclass: #Hero
  instanceVariableNames: 'name level powers'
  classVariableNames: ''
  package: 'SuperHeroes'
```

```
Hero >> name
  ^ name

Hero >> name: aString
  name := aString
```

```
Hero >> level
  ^ level

Hero >> level: anObject
  level := anObject
```

```
Hero >> powers
  ^ powers ifNil: [ powers := Set new ]
```

```
Hero >> addPower: aPower
  self powers add: aPower
```

... and Powers

Let us define the class Power.

```
Object subclass: #Power
  instanceVariableNames: 'name'
  classVariableNames: ''
  package: 'SuperHeroes'
```

```
Power >> name
  ^ name

Power >> name: aString
  name := aString
```

Ajoutez les méthodes printOn: afin d'améliorer la navigation et le débuggage de vos super heroes.

1.4 Root classes

Now we have to decide what are the objects that we want to save and query. For this we should declare the roots of the object graph that we want to save. A root can be any class of the system. Declaring a root is done by implementing the class method isVoyageRoot on the class of the objects that we want to save. We will see the implications of defining a root later. For now we just define Hero as root.

```
Hero class >> isVoyageRoot
  ^ true
```

We can create some superheroes and save them in the database.

```
Hero new
  name: 'Spiderman';
  level: #epic;
  addPower: (Power new name: 'Super-strength');
  addPower: (Power new name: 'Wall-climbing');
  addPower: (Power new name: 'Spider instinct');
  save.
Hero new
  name: 'Wolverine';
  level: #epic;
  addPower: (Power new name: 'Regeneration');
  addPower: (Power new name: 'Adamantium claws');
  save.
```

1.5 Checking in MongoDB

We can check directly in the database to see how our objects are saved.

```
> show dbs
local       0.078GB
superHeroes  0.078GB

> use superHeroes
switched to db superHeroes

> show collections
Hero
```

Now we can see how a superhero is actually stored. db.Hero.find()[0] gets the first object of the collection.

```
> db.Hero.find()[0]
{
    "_id" : ObjectId("d847065c56d0ad09b4000001"),
    "#version" : 688076276,
    "#instanceOf" : "Hero",
    "level" : "epic",
    "name" : "Spiderman",
    "powers" : [
        {
            "#instanceOf" : "Power",
            "name" : "Spider instinct"
        },
        {
            "#instanceOf" : "Power",
            "name" : "Super-strength"
```

```
      },
      {
         "#instanceOf" : "Power",
         "name" : "Wall-climbing"
      }
   ]
}
```

Note the way the powers are saved: they are embedded inside the document that represents the superhero.

1.6 Queries

Now from Pharo, we can perform some queries to get objects stored in the database.

```
Hero selectAll.
> an OrderedCollection(a Hero( Spiderman ) a Hero( Wolverine )
```

```
Hero selectOne: [ :each | each name = 'Spiderman' ].
> a Hero( Spiderman )
```

```
Hero selectMany: [ :each | each level = #epic ].
> an OrderedCollection(a Hero( Spiderman ) a Hero( Wolverine )
```

Since MongoDB is storing internally JSON, the argument of a query can be a dictionary as follows:

```
Hero selectOne: { #name -> 'Spiderman' } asDictionary.
> a Hero( Spiderman )
```

```
Hero selectMany: { #level -> #epic } asDictionary.
> an OrderedCollection(a Hero( Spiderman ) a Hero( Wolverine )
```

Here is a more complex query:

```
Hero
   selectMany: { #level -> #epic } asDictionary
   sortBy: { #name -> VOOrder ascending } asDictionary
   limit: 10
   offset: 0
```

1.7 Other Basic Operations

Here are some simple operations that can be performed on root classes.

Counting

First we show how we can count:

```
Hero count.
> 2
```

```
Hero count: [ :each | each name = 'Spiderman' ]
> 1
```

Removing

We can remove objects from the database.

```
hero := Hero selectAll anyOne.
hero remove.
> a Hero
```

We can also remove all the objects from the class.

```
Hero removeAll. Beware of this!
> Hero class
```

1.8 Adding a new root

Now we will change our requirement and show that we want to be able to query another class of objects: the powers. Note that when you add a root, it is important that you either flush your database or perform a migration by for example loading old objects are republishing them.

Each time you change the database 'schema', you should reset the database using the following expression:

```
VORepository current reset.
```

When to add a new root

There are two main points to consider when facing the questions of the necessity of adding a class as a root.

- First, the obvious consideration is whether we need to query objects separately from their objects that refer to them.

- Second, if you need to make sure that subparts will be shared and not duplicated you should declare the subparts as root. For example if you need to be able to share a power between two super heroes and want to be sure that when you load the two superheroes you do not get two copies of the same power.

Power as a root

We declare Power as a new root.

```
Power class >> isVoyageRoot
   ^ true
```

Now we can save the super power objects separately as follows:

```
Power new name: 'Fly'; save.
Power new name: 'Super−strength'; save.
```

If you do not see the new collection in the database using show collections you may face a Voyage bug and you need to reset the memory database cache in the Pharo image doing:

```
VORepository current reset.
```

Now saving your objects and checking the mongo db again should show

```
> show collections
   Hero
   Power
```

Now we can save a hero and its superpowers. To fully test we flush the heroes in the database executing Hero removeAll and we execute the following:

```
| fly superStrength |
fly := Power selectOne: [ :each | each name = 'Fly'].
superStrength := Power selectOne: [ :each | each name = 'Super−strength'].
Hero new
   name: 'Superman'; level: #epic;
   addPower: fly;
   addPower: superStrength;
   save.
```

Note that while we saved the powers independently from the hero, this is not mandatory since saving a hero will automatically save its powers.

Now when we query the database we can see that an hero has references to another collection of Powers and that the powers are not nested inside the hero documents.

```
> db.Hero.find()[0]
{
    "_id" : ObjectId("d8474983421aa909b4000008"),
    "#version" : NumberLong("3874503784"),
    "#instanceOf" : "Hero",
    "level" : "epic",
    "name" : "Superman",
    "powers" : [
        {
            "#collection" : "Power",
            "#instanceOf" : "Power",
            "__id" : ObjectId("d84745dd421aa909b4000005")
        },
        {
            "#collection" : "Power",
            "#instanceOf" : "Power",
            "__id" : ObjectId("d84745dd421aa909b4000006")
        }
    ]
}
```

About relations

Voyage supports cyclic references between root objects but it does not support cyclic references to embedded objects. We will see that in the following section.

Extending the Hero class

We will now extend the class Hero with equipments. This example shows that the root collection declaration is *static*: when a superclass is defined as root, the collection in the mongo db will contain instances of both the class and its subclasses. If we want to have a collection per subclass we have to define each of them as root and you should duplicate the isVoyageRoot method in each class.

We add a new instance variable named equipment to the class Hero.

```
Object subclass: #Hero
  instanceVariableNames: 'name level powers equipment'
  classVariableNames: ''
  package: 'SuperHeroes'
```

```
Hero >> equipment
  ^ equipment ifNil: [ equipment := Set new ]
```

```
Hero >> addEquipment: anEquipment
  self equipment add: anEquipment
```

Since we change the class structure we should reset the local cache of the database doing VORepository current reset.

Now we define the class Equipment as a new root.

```
Object subclass: #Equipment
   instanceVariableNames: ''
   classVariableNames: ''
   package: SuperHeroes'
```

```
Equipment class >> isVoyageRoot
   ^ true
```

And we define two subclasses for Weapon and Armor

```
Equipment subclass: #Weapon
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'SuperHeroes'
```

```
Equipment subclass: #Armor
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'SuperHeroes'
```

Now saving a new hero with equipment will also save its equipment as a separate object.

```
Hero new
   name: 'Iron−Man';
   level: #epic;
   addEquipment: Armor new;
   save.
```

We can see how the objects are saved in the database

```
> db.Hero.find()[1]
{
   "_id" : ObjectId("d8475734421aa909b4000001"),
   "#instanceOf" : "Hero",
   "#version" : NumberLong("2898020230"),
   "equipment" : [
      {
         "#instanceOf" : "Armor"
      }
```

```
    ],
    "level" : "epic",
    "name" : "Iron-Man",
    "powers" : null
}
```

Since we did not define Weapon and Armor has separate roots, there is only one collection named Equipment in the database containing both weapons and armors.

Equipment can also have powers

In fact equipments can also have powers (like the hammer of Thor). Therefore we add powers to the equipments as follows:

```
Object subclass: #Equipment
    instanceVariableNames: 'powers'
    classVariableNames: ''
    package: 'SuperPowers'
```

```
Equipment >> powers
    ^ powers ifNil: [ powers := Set new ]

Equipment >> addPower: aPower
    self powers add: aPower
```

Since we change the class structure we should reset the local cache of the database doing

```
VORepository current reset
```

And we can now add a equipment with powers to Ironman as follows:

```
| hero fly superStrength |
hero := Hero selectOne: [ :each | each name = 'Iron-Man' ].
fly := Power selectOne: [ :each | each name = 'Fly' ].
superStrength := Power selectOne: [ :each | each name = 'Super-strength' ].
hero addEquipment: (Armor new
        addPower: fly;
        addPower: superStrength;
        yourself);
save.
```

We see in the database that the Equipment collection contains Armor objects.

```
> db.Equipment.find()[0]
```

```
{
   "_id" : ObjectId("d8475777421aa909b4000003"),
   "#instanceOf" : "Armor",
   "#version" : NumberLong("4204064627")
}
```

Note that an equipment could contain an equipment. To express this we do not have anything to handle cyclic references since the class Equipment is a collection root.

1.9 Conclusion

This little tutorial shows how easy it is to store objects in a Mongo database. It complements the space of possible solutions such as using Fuel to serialize object, using the in-memory SandStone approach or the more traditional relation database mapping with Garage.