Chapter 1

Persisting Object with Voyage

Voyage is a small persistence framework developed by E. Lorenzano. It is purely object-oriented and with the goal to present a common API to most common development usages. It is a small layer between your objects and the persistent mechanism (such a Mongo database). This layer provides some useful vocabulary for your objects.

1.1 Getting Started

Load

To install Voyage you can go to Configurations Browser (in World Menu/Tools) and load ConfigurationOfVoyageMongo. Or alternatively you can execute in your workspace:

```
Gofer it
   url: 'http://smalltalkhub.com/mc/estebanlm/Voyage/main';
   package: 'ConfigurationOfVoyageMongo';
   load.
(Smalltalk at: #ConfigurationOfVoyageMongo) load.
```

That will load all you need to persist your object model into a Mongo database.

Create a repository

Start a new repository or connect with an existing one is very simple, you just need to do:

```
repository := VOMongoRepository
   host: 'mongo.db.url'
   database: 'databaseName'.
```

Alternatively you can specify the port to connect by using protocol VOMongoRepository class>>host:port:database:. You can keep that instance for storage/retrieval, but probably you prefer to use the Singleton mode (See explanation below). To enable your repository to be accessed all around the image, you can execute:

```
repository enableSingleton.
```

WARNING: executing this line will remove older repositories from Singleton mode!

NOTE: In this document, we are going to cover Voyage in Singleton mode, but using it in Instance mode should be straightforward (just take a look at VORepository persistence protocol).

1.2 Voyage

Voyage can work in two different programming styles (modes):

- Singleton mode: You have an unique repository in your image, which works as a singleton keeping all your data there. When you use this style, you can program using a "behavioral complete" approach: your instances respond to a certain vocabulary (see below to more details about vocabulary and usage).

- Instance mode: You can have an undetermined number of repositories living in your image. Of course, this mode requires you to make explicit which repository are you going to use.

Voyage API

The following table shows the complete API of Voyage

Singleton mode	Instance mode	Description
save	save:	stores an object into repository (insert or u
remove	remove:	removes an object from repository
removeAll	removeAll:	removes all objects of class from reposit
selectAll	selectAll:	retrieves all objects of some kind
selectOne:	selectOne:where:	retrieves first object that matches the where
selectMany:	selectMany:where:	retrieves all objects that matches the where

Voyage Mongo Back-End

Voyage is a common layer for different backends but currently it supports just two: an *in memory* layer (to fast prototype applications and to fist stage developments), and a Mongo database backend.

Mongo is a document-oriented database that works well persisting complex object models, even if it is not an object database (like Gemstone, Magma or Omnibase). Because of that, we need an Object-Document mapper (the document equivalent to an ORM), and while it does not solve all the impedance mismatch issues, it fits a lot better with an object world, being able to preserve it's dynamic nature.

1.3 Storing objects

Basic storage

Let's say we want to store an Association (a pair of objects). To do that, we need to declare that the class Association is storable as root (first entrance point) of our repository. To express this we define the method isVoyageRoot to return true.

```
Association class>>isVoyageRoot
  ^true
```

Then, you just need to execute:

```
anAssociation := #answer->42.
anAssociation save.
```

This will generate a collection in your database with a document having the following structure.

```
{
  "_id" : ObjectId("a05feb630000000000000000"),
  "#instanceOf" : "Association",
  "#version" : NumberLong("3515916499"),
```

```
   "key" : 'answer',
   "value" : 42
}
```

Such collection in the database is named "point".

Note: Stef: why named Point??

As you can see, there are some "extra information" to allow the object to be recognized:

- instanceOf keeps the type (Class) of the stored instance. This is useful for reconstructing complex hierarchies.
- version keeps a marker of the version committed. This property is used for refreshing cached data in the application. If there is not version field the application will always refresh the object with data from database (and you will lose a lot of performance).

Embedding objects

Now, most objects you use are not so simple as associations, but more complex graphs with other complex object instances inside. Let's say that we want to store rectangles. Each rectangle contains two points.

To store this, we specify again that the Rectangle class can be a document root as follows:

```
Rectangle class>>isVoyageRoot
   ^true
```

Now we can save rectangles as shown by this snippet:

```
aRectangle := 42@1 corner: 10@20.
aRectangle save.
```

This will generate a document like this:

```
{
   "_id" : ObjectId("ef72b58100000000000000000"),
   "#instanceOf" : "Rectangle",
   "#version" : NumberLong("2460645040"),
   "origin" : {
      "#instanceOf" : "Point",
      "x" : 42,
      "y" : 1
```

```
   },
   "corner" : {
      "#instanceOf" : "Point",
      "x" : 10,
      "y" : 20
   }
}
```

As you can see, for many cases, you do not need to do anything special to store complex graphs (but you can enhance and/or modify the way you persist objects, see Section 1.3 below).

Referencing other roots

Sometimes your objects are graphs that contain other root objects. For instance, you could want to keep users and roles in different collections, and of course, an user has a collection of roles. To do that, you just need to declare the former embedded objects as roots.

In our rectangle example, let's suppose we want to keep into a separated collection the points (we call them referenced instead embedded).

After we add isVoyageRoot to Point class, we can save our rectangle.

```
Point class>>isVoyageRoot
   ^ true
```

We get in the collection *rectangle* the following entities:

```
{
   "_id" : ObjectId("7c5e772b0000000000000000"),
   "#instanceOf" : "Rectangle",
   "#version" : 423858205,
   "origin" : {
      "#collection" : "point",
      "#instanceOf" : "Point",
      "__id" : ObjectId("7804c56c0000000000000000")
   },
   "corner" : {
      "#collection" : "point",
      "#instanceOf" : "Point",
      "__id" : ObjectId("2a731f310000000000000000")
   }
}
```

In addition we also get in the collection *point* the two documents (entities):

```
{
```

```
    "_id" : ObjectId("7804c56c0000000000000000"),
    "#version" : NumberLong("4212049275"),
    "#instanceOf" : "Point",
    "x" : 42,
    "y" : 1
}

{
    "_id" : ObjectId("2a731f310000000000000000"),
    "#version" : 821387165,
    "#instanceOf" : "Point",
    "x" : 10,
    "y" : 20
}
```

Enhancing storage

You can change the way your objects are stored by adding Magritte descriptions.

For instance, always with our rectangle example but with points embedded and not referenced, let's image the following constraints:

- We want to use a different collection named *rectanglesForTest* instead *rectangle*.
- We know that we will store always points in my collection, and therefore the instanceOf information is redundant.
- We know that *origin* and *corner* attributes are going to be always points, so instanceOf information is redundant there too.

We will use specific pragmas to declare such properties.

We defined three class side methods, mongoContainer, mongoOrigin and mongoCorner.

The method mongoContainer is defined as follows: It uses the <mongoContainer> pragma. It sets the the name of the collection and the kind of entity (here Rectangle).

```
Rectangle class>>mongoContainer
  <mongoContainer>

  ^VOMongoContainer new
    collectionName: 'rectanglesForTest';
    kind: Rectangle;
    yourself
```

The two other methods use the pragma <mongoDescription>. They set their respective attribute name and kind.

```
Rectangle class>>mongoOrigin
   <mongoDescription>

   ^VOMongoToOneDescription new
      attributeName: 'origin';
      kind: Point;
      yourself
```

```
Rectangle class>>mongoCorner
   <mongoDescription>

   ^VOMongoToOneDescription new
      attributeName: 'corner';
      kind: Point;
      yourself
```

Then, you can save your rectangle and you will see that the document generated (now in *rectanglesForTest*), it will look more or less like this:

```
{
   "_id" : ObjectId("ef72b58100000000000000000"),
   "#version" : NumberLong("2460645040"),
   "origin" : {
      "x" : 42,
      "y" : 1
   },
   "corner" : {
      "x" : 10,
      "y" : 20
   }
}
```

You can do other fancy configurations, like the following one

- Using the message beEager can declare that the referenced instance to be loaded eagerly (default is lazy).
- Using the message beLazy can declare referenced instances to be loaded lazily.
- Using the message kindCollection:. For VOMongoToManyDescription, it allows you to change the resulting collection kind.
- Using the message convertNullTo:. When you are retrieving an object (or a collection of objects) which value is Null (nil), you can

instead evaluate the valuable you are passing as an argument.

1.4 Conclusion

This is just the start of the chapter.