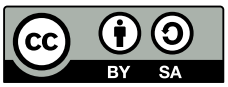
Voyage: Persisting Objects in Document Databases

Esteban Lorenzano, Stéphane Ducasse, Johan Fabry and Norbert Hartl

The Pharo Booklet Collection — edited by S. Ducasse
September 14, 2017
master @ 977d96c



Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Voyage

Voyage is a small persistence framework developed by Esteban Lorenzano, constructed as a small layer between the objects and a persistency mechanism often a document noSql database.

This booklet started as a number of blog posts by Esteban Lorenzano, which have been extensively reworked by Johan Fabry and Stéphane Ducasse, including additional information shared by Sabine Manaa and Norbert Hartl. This became the chapter in the Enterprise Pharo book available at *http://books.pharo.org>http://books.pharo.org). Since this chapter was complex to edit without producing a complete version of the book and that extra material such as the super heroes tutorial written by Stephane Ducasse appeared the current booklet is a merge of all the sources and will be the most actively maintained documentation.

1.1 What is Voyage?

It is purely object-oriented and has as a goal to present a minimal API to most common development usages. Voyage is a common layer for different backends but currently it supports just two: an *in-memory* layer and a backend for the MongoDB database (http://mongodb.org[1]) and UnqLite (https://www.unqlite.org).

The in-memory layer is useful to prototype applications quickly and for initial development without a database back-end, for example using the Pharo image as the persistency mechanism.

The MongoDB database backend stores the objects in a document-oriented database. In MongoDB each stored entity is a JSON-style document. This document-centric nature allows for persisting complex object models in a fairly straightforward fashion. MongoDB is not an object database, like Gemstone, Magma or Omnibase, so there still is a small gap to be bridged between objects and documents. To bridge this gap, Voyage contains a mapper converting objects to and from documents. This mapper is equivalent to an Object-Relational Mapper (ORM) when using relational databases. While this mapper does not solve all the known impedance mismatch issues when going from objects to a database, we find that using a document database fits better with the object world than a combination of a ORM and a relational database. This is because document databases tend to provide better support for the dynamic nature of the object world.

Voyage provides a default way in which objects are stored in the database. Fine-grained configuration of this can be performed using Magritte descriptions. Voyage also includes a query API, which allows specific objects to be retrieved from a MongoDB database. We will discuss each of these features in this text.

[1] http://mongodb.org/

1.2 Voyage vision

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.

1.3 Contents

This booklet has several chapters

- One is a simple tutorial to get started with Voyage.
- Then a more complete overview of the API is described.
- Finally a chapter gathering tips and tricks is presented.

1.4 Load Voyage

To install Voyage, including support for the MongoDB database, go to the Configurations Browser (in the World Menu/Tools) and load ConfigurationOfVoyageMongo. Or alternatively execute in a workspace:

```
Gofer it
   url: 'http://smalltalkhub.com/mc/estebanlm/Voyage/main';
   configurationOf: 'VoyageMongo';
   loadStable.
```

This will load all that is needed to persist objects into a Mongo database.

1.5 Install your document databases

MongoDB

Next is to install the MongoDB database. How to do this depends on the operating system, and is outside of the scope of this text. We refer to the MongoDB website[2] for more information.

[2] http://www.mongodb.org/downloads

CHAPTER 2

A Simple Tutorial with Super Heroes

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.

2.1 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 |
  repository := VOMongoRepository
    host: 'localhost'
    database: 'superHeroes'.
  repository enableSingleton.
```

2.2 SuperHeroes

Now we can define a first version of our domain. Figure 2-1 shows the model that we will use for this tutorial.

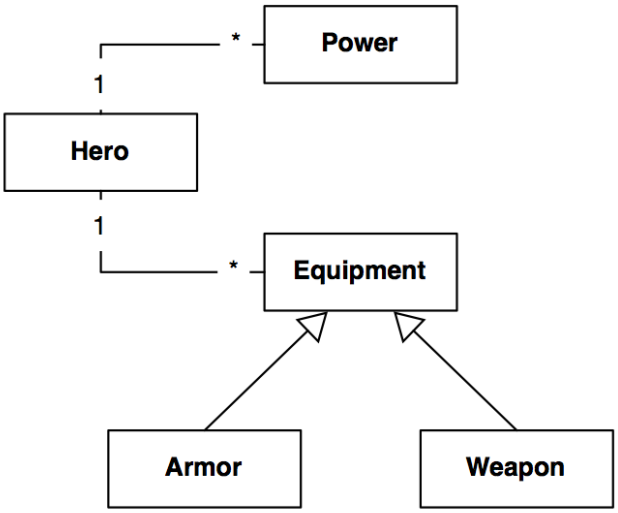


Figure 2-1 The model: SuperHeroes, SuperPowers and their Equipments.

2.3 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
```

2.4 ... 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.

2.5 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.
```

2.6 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.

2.7 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
```

2.8 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
```

2.9 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.

2.10 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")
    }
  ]
}
```

2.11 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.

2.12 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.

2.13 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.

2.14 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.

CHAPTER 3

Persisting Objects with Voyage

In this chapter we will do a tour of Voyage API.

3.1 Create a repository

In Voyage, all persistent objects are stored in a repository. The kind of repository that is used determines the storage backend for the objects.

To use the in-memory layer for Voyage, an instance of `VOMemoryRepository` needs to be created, as follows:

```
repository := VOMemoryRepository new
```

In this text, we shall however use the MongoDB backend. To start a new MongoDB repository or connect to an existing repository create an instance of `VOMongoRepository`, giving as parameters the hostname and database name. For example, to connect to the database `databaseName` on the host `mongo.db.url` execute the following code:

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

Alternatively, using the message `host:port:database:` allows to specify the port to connect to. Lastly, if authentication is required, this can be done using the message `host:database:username:password:` or the message `host:port:database:username:password:`.

3.2 Singleton mode and instance mode

Voyage can work in two different modes:

- Singleton mode: There is an unique repository in the image, which works as a singleton keeping all the data. When you use this mode, you can program using a ”behavioral complete” approach where instances respond to a certain vocabulary (see below for more details about vocabulary and usage).
- Instance mode: You can have an undetermined number of repositories living in the image. Of course, this mode requires you to make explicit which repositories you are going to use.

By default, Voyage works in instance mode: the returned instance has to be passed as an argument to all database API operations. Instead of having to keep this instance around, a convenient alternative is to use Singleton mode. Singleton mode removes the need to pass the repository as an argument to all database operations. To use Singleton mode, execute:

```
repository enableSingleton.
```

> **Note** Only one repository can be the singleton, hence executing this line will remove any other existing repositories from Singleton mode! In this document, we cover Voyage in Singleton mode, but using it in Instance mode is straightforward as well. See the protocol `persistence` of `VORepository` for more information.

3.3 Voyage API

The following two tables show a representative subset of the API of Voyage. These methods are defined on `Object` and `Class`, but will only truly perform work if (instances of) the receiver of the message is a Voyage root. See the `voyage-model-core-extensions` persistence protocol on both classes for the full API of Voyage.

First we show Singleton mode:

`save`	stores an object into repository (insert or update)
`remove`	removes an object from repository
`removeAll`	removes all objects of class from repository
`selectAll`	retrieves all objects of some kind
`selectOne:`	retrieves first object that matches the argument
`selectMany:`	retrieves all objects that matches the argument

Second is Instance mode. In Instance mode, the first argument is always the repository on which to perform the operation.

`save:`	stores an object into repository (insert or update)
`remove:`	removes an object from repository
`removeAll:`	removes all objects of class from repository
`selectAll:`	retrieves all objects of some kind
`selectOne:where:`	retrieves first object that matches the where clause
`selectMany:where:`	retrieves all objects that matches the where clause

3.4 Resetting or dropping the database connection

In a deployed application, there should be no need to close or reset the connection to the database. Also, Voyage re-establishes the connection when the image is closed and later reopened.

However, when developing, resetting the connection to the database may be needed to reflect changes. This is foremost required when changing storage options of the database (see section 3.11). Performing a reset is achieved as follows:

```
VORepository current reset.
```

In case the connection to the database needs to be dropped, this is performed as follows:

```
VORepository setRepository: nil.
```

3.5 Storing objects

To store objects, the class of the object needs to be declared as being a *root of the repository*. All repository roots are points of entry to the database. Voyage stores more than just objects that contain literals. Complete trees of objects can be stored with Voyage as well, and this is done transparently. In other words, there is no need for a special treatment to store trees of objects. However, when a graph of objects is stored, care must be taken to break loops. In this section we discuss such basic storage of objects, and in section 3.11 on Enhancing Storage we show how to enhance and/or modify the way objects are persisted.

3.6 Basic storage

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

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

We can also define the name of the collection that will be used to store documents with the `voyageCollectionName` class method. By default, Voyage creates a MongoDB collection for each root class with name the name of the class.

```
Association class>>voyageCollectionName
  ^ 'Associations'
```

Then, to save an association, we need to just send it the `save` message:

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

This will generate a collection in the database containing a document of the following structure:

```
{
  "_id" : ObjectId("a05feb630000000000000000"),
  "#instanceOf" : "Association",
  "#version" : NumberLong("3515916499"),
  "key" : 'answer',
  "value" : 42
}
```

The stored data keeps some *extra information* to allow the object to be correctly reconstructed when loading:

- `instanceOf` records the class of the stored instance. This information is important because the collection can contain subclass instances of the Voyage root class.
- `version` keeps a marker of the object version that is committed. This property is used internally by Voyage for refreshing cached data in the application. Without a `version` field, the application would have to refresh the object by frequently querying the database.

Note that the documents generated by Voyage are not directly visible using Voyage itself, as the goal of Voyage is to abstract away from the document structure. To see the actual documents you need to access the database directly. For MongoDB this can be done through Mongo Browser, which is loaded as part of Voyage (World->Tools->Mongo Browser). Other options for MongoDB are to use the `mongo` command line interface or a GUI tool such as RoboMongo[1] (Multi-Platform) or MongoHub[2] (for Mac).

3.7 Embedding objects

Objects can be as simple as associations of literals or more complex: objects can contain other objects, leading to a tree of objects. Saving such objects is as simple as sending the `save` message to them. For example, let's say that we want to store rectangles and that each rectangle contains two points. To achieve this, we specify that the `Rectangle` class is a document root as follows:

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

This allows rectangles to be saved to the database, for example as shown by this snippet:

[1] http://robomongo.org
[2] http://mongohub.todayclose.com/

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

This will add a document to the `rectangle` collection of the database with this structure:

```
{
  "_id" : ObjectId("ef72b5810000000000000000"),
  "#instanceOf" : "Rectangle",
  "#version" : NumberLong("2460645040"),
  "origin" : {
    "#instanceOf" : "Point",
    "x" : 42,
    "y" : 1
  },
  "corner" : {
    "#instanceOf" : "Point",
    "x" : 10,
    "y" : 20
  }
}
```

3.8 Referencing other roots

Sometimes the objects are trees that contain other root objects. For instance, you could want to keep users and roles as roots, i.e. in different collections, and a user has a collection of roles. If the embedded objects (the roles) are root objects, Voyage will store references to these objects instead of including them in the document.

Returning to our rectangle example, let's suppose we want to keep the points in a separate collection. In other words, now the points will be referenced instead of embedded.

After we add `isVoyageRoot` to `Point class`, and save the rectangle, in the `rectangle` collection, we get the following document:

```
 {
  "_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 to this, in the collection `point` we also get the two following entities:

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

{
  "_id" : ObjectId("2a731f310000000000000000"),
```

```
  "#version" : 821387165,
  "#instanceOf" : "Point",
  "x" : 10,
  "y" : 20
}
```

3.9 Breaking cycles in graphs

When the objects to be stored contain a graph of embedded objects instead of a tree, i.e. when there are cycles in the references that the embedded objects have between them, the cycles between these embedded objects must be broken. If not, storing the objects will cause an infinite loop. The most straightforward solution is to declare one of the objects causing the cycle as a Voyage root. This effectively breaks the cycle at storage time, avoiding the infinite loop.

For example, in the rectangle example say we have a label inside the rectangle, and this label contains a piece of text. The text also keeps a reference to the label in which it is contained. In other words there is a cycle of references between the label and the text. This cycle must be broken in order to persist the rectangle. To do this, either the label or the text must be declared as a Voyage root.

An alternative solution to break cycles, avoiding the declaration of new voyage roots, is to declare some fields of objects as transient and define how the graph must be reconstructed at load time. This will be discussed in the following section.

3.10 Storing instances of Date in Mongo

A known issue of mongo is that it does not make a difference between `Date` and `DateAndTime`, so even if you store a `Date` instance, you will retrieve a `DateAndTime` instance. You will have to transform it back to `Date` manually when materializing the object.

3.11 Enhancing storage

How objects are stored can be changed by adding Magritte descriptions to their classes. In this section, we first talk about configuration options for the storage format of the objects. Then we treat more advanced concepts such as loading and saving of attributes, which can be used, for example, to break cycles in embedded objects.

Configuring storage

Consider that, continuing with the rectangle example but using embedded points, we add the following storage requirements:

- We need to use a different collection named `rectanglesForTest` instead of `rectangle`.
- We only store instances of the `Rectangle` class in this collection, and therefore the `instanceOf` information is redundant.
- The `origin` and `corner` attributes are always going to be points, so the `instanceOf` information there is redundant as well.

To implement this, we use Magritte descriptions with specific pragmas to declare properties of a class and to describe both the `origin` and `corner` attributes.

The method `mongoContainer` is defined as follows: First it uses the pragma `<mongoContainer>` to state that it describes the container to be used for this class. Second it returns a specific `VOMongoContainer` instance. This instance is configured such that it uses the `rectanglesForTest` collection in the database, and that it will only store `Rectangle` instances.

Note that it is not required to specify both configuration lines. It is equally valid to only declare that the collection to be used is `rectanglesForTest`, or only specify that the collection contains just `Rectangle` instances.

```
Rectangle class>>mongoContainer
  <mongoContainer>

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

The two other methods use the pragma `<mongoDescription>` and return a Mongo description that is configured with their respective attribute name and kind, as follows:

```
Rectangle class>>mongoOrigin
  <mongoDescription>

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

```
Rectangle class>>mongoCorner
  <mongoDescription>

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

After resetting the repository with:

```
VORepository current reset
```

a saved rectangle, now in the `rectanglesForTest` collection, will look more or less as follows:

```
{
  "_id" : ObjectId("ef72b5810000000000000000"),
  "#version" : NumberLong("2460645040"),
  "origin" : {
    "x" : 42,
    "y" : 1
  },
  "corner" : {
    "x" : 10,
    "y" : 20
  }
}
```

Other configuration options for attribute descriptions are:

- `beEager` declares that the referenced instance is to be loaded eagerly (the default is lazy).
- `beLazy` declares that referenced instances are loaded lazily.
- `convertNullTo:` when retrieving an object whose value is Null (`nil`), instead return the result of evaluating the block passed as argument.

For attributes which are collections, the `VOMongoToManyDescription` needs to be returned instead of the `VOMongoToOneDescription`. All the above configuration options remain valid, and the `kind:` configuration option is used to specify the kind of values the collection contains.

`VOMongoToManyDescription` provides a number of extra configuration options:

- `kindCollection:` specifies the class of the collection that is contained in the attribute.
- `convertNullToEmpty` when retrieving a collection whose value is Null (`nil`), it returns an empty collection.

3.12 Custom loading and saving of attributes

It is possible to write specific logic for transforming attributes of an object when written to the database, as well as when read from the database. This can be used, e.g., to break cycles in the object graph without needing to declare extra Voyage roots. To declare such custom logic, a `MAPluggableAccessor` needs to be defined that contains Smalltalk blocks for reading the attribute from the object and writing it to the object. Note that the names of these accessors can be counter-intuitive: the `read:` accessor defines the value that will be **stored** in the database, and the `write:` accessor defines the transformation of this **retrieved** value to what is placed in the object. This is because the accessors are used by the Object-Document mapper when **reading the object** to store it to the database and when **writing the object** to memory, based on the values obtained from the database.

Defining accessors allows, for example, a `Currency` object that is contained in an `Amount` to be written to the database as its' three letter abbreviation (EUR, USD, CLP, ...). When loading this representation, it needs to be converted back into a Currency object, e.g. by instantiating a new Currency object. This is achieved as follows:

```
Amount class>>mongoCurrency
  <mongoDescription>

  ^ VOMongoToOneDescription new
    attributeName: 'currency';
    accessor: (MAPluggableAccessor
      read: [ :amount | amount currency abbreviation ]
      write: [ :amount :value | amount currency: (Currency fromAbbreviation: value) ]);
    yourself
```

Also, a post-load action can be defined for an attribute or for the containing object, by adding a `postLoad:` action to the attribute descriptor or the container descriptor. This action is a one-parameter block, and will be executed after the object has been loaded into memory with as argument the object that was loaded.

Lastly, attributes can be excluded from storage (and hence retrieval) by returning a `VOMongoTransientDescription` instance as the attribute descriptor. This allows to place cut-off points in the graph of objects that is being saved, i.e. when an object contains a reference to data that should not be persisted in the database. This may also be used to break cycles in the stored object graph. It however entails that when retrieving the graph from the database, attributes that contain these objects will be set to `nil`. To address this, a post-load action can be specified for the attribute descriptor or the container descriptor, to set these attributes to the correct values.

3.13 A few words concerning the OID

The mongo ObjectId (OID) is a unique field acting as a primary key. It is a 12-byte BSON type, constructed using:

- a 4-byte value representing seconds passed since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id,
- a 3-byte counter, starting with a random value.

Objects which are added into a mongo root collection get a unique id, instance of `OID`. If you create such an object and then ask it for its OID by sending it `voyageId`, you get the OID. The instance variable `value` of the OID contains a `LargePositiveInteger` that corresponds to the mongo ObjectId.

It is possible to create and use your own implementation of OIDs and put these objects into the mongo database. But this is not recommended as you possibly may no longer be able to query these objects by their OID (by using `voyageId`), since mongo expects a certain format. If you do, you should check your format by querying for it in the mongo console, for example as below. If you get the `result Error: invalid object id: length`, then you will not be able to query this object by id.

```
> db.Trips.find({"person._id" : ObjectId("190372")})
Fri Aug 28 14:21:10.815 Error: invalid object id: length
```

An extra advantage of the OID in the mongo format is that these are ordered by creation date and time and as a result you have an indexed "creationDateAndTime" attribute for free (since there is a non deletable index on the field of the OID `_id`).

3.14 Querying in Voyage

Voyage allows to selectively retrieve object instances though queries on the database. When using the in-memory layer, queries are standard Smalltalk blocks. When using the MongoDB back-end, the MongoDB query language is used to perform the searches. To specify these queries, MongoDB uses JSON structures, and when using Voyage there are two ways in which these can be constructed. MongoDB queries can be written either as blocks or as dictionaries, depending on their complexity. In this section, we first discuss both ways in which queries can be created, and we end the section by talking about how to execute these queries.

3.15 Basic object retrieval using blocks or mongoQueries

The most straightforward way to query the database is by using blocks when using the in-memory layer or MongoQueries when using the MongoDB back-end. In this discussion we will focus on the use of MongoQueries, as the use of blocks is standard Smalltalk.

MongoQueries is not part of Voyage itself but part of the MongoTalk layer that Voyage uses to talk to MongoDB. MongoTalk was made by Nicolas Petton and provides all the low-level operations for accessing MongoDB. MongoQueries transforms, within certain restrictions, regular Pharo blocks into JSON queries that comply to the form that is expected by the database. In essence, MongoQueries is an embedded Domain Specific Language to create MongoDB queries. Using MongoQueries, a query looks like a normal Pharo expression (but the language is much more restricted than plain Smalltalk).

Using MongoQueries, the following operators may be used in a query:

`< <= > >= = ~=`	Regular comparison operators
`&`	AND operator
`|`	OR operator
`not`	NOT operator
`at:`	Access an embedded document
`where:`	Execute a Javascript query

For example, a query that selects all elements in the database whose name is `John` is the following:

```
[ :each | each name = 'John' ]
```

A slightly more complicated query is to find all elements in the database whose name is `John` and the value in `orders` is greater than 10.

```
[ :each | (each name = 'John') & (each orders > 10 ) ]
```

Note that this way of querying only works for querying values of the object but not values of references to other objects. For such case you should build your query using ids, as traditionally done in relational database, which we talk about next. However the best solution in the Mongo spirit of things is to revisit the object model to avoid relationships that are expressed with foreign keys.

3.16 Quering with elements from another root document

With No-SQL databases, it is impossible to query on multiple collections (the equivalent of a JOIN statement in SQL). You have two options: alter your schema, as suggested above, or write application-level code to reproduce the JOIN behavior. The latter option can be done by sending the `voyageId` message to an object already returned by a previous query and using that id to match another object. An example where we match colors `color` to a reference color `refCol` is as follows:

```
[ :each | (each at: 'color._id') = refCol voyageId ]
```

3.17 Using the at: message to access embedded documents

Since MongoDB stores documents of any complexity, it is common that one document is composed of several embedded documents, for example:

```
{
  "origin" : {
    "x" : 42,
    "y" : 1
  },
  "corner" : {
    "x" : 10,
    "y" : 20
  }
}
```

In this case, to search for objects by one of the embedded document elements, the message `at:`, and the field separator **".**" needs to be used. For example, to select all the rectangles whose origin x value is equal to 42, the query is as as follows.

```
[ :each | (each at: 'origin.x') = 42 ]
```

3.18 Using the where: message to perform Javascript comparisons

To perform queries which are outside the capabilities of MongoQueries or even the MongoDB query language, MongoDB provides a way to write queries directly in Javascript using the `$where` operand. This is also possible in MongoQueries by sending the `where:` message:

In the following example we repeat the previous query with a Javascript expression:

```
[ :each | each where: 'this.origin.x == 42' ].
```

More complete documentation about the use of `$where` is in the MongoDB where documentation[3].

3.19 Using JSON queries

When MongoQueries is not powerful enough to express your query, you can use a JSON query instead. JSON queries are the MongoDB query internal representation, and can be created straightforwardly in Voyage. In a nutshell: a JSON structure is mapped to a dictionary with pairs. In these

[3] http://docs.mongodb.org/manual/reference/operator/where/#op._S_where

pairs the key is a string and the value can be a primitive value, a collection or another JSON structure (i.e., another dictionary). To create a query, we simply need to create a dictionary that satisfies these requirements.

> **Note** The use of JSON queries is strictly for when using the MongoDB back-end. Other back-ends, e.g., the in-memory layer, do not provide support for the use of JSON queries.

For example, the first example of the use of MongoQueries is written as a dictionary as follows:

```
{ 'name' -> 'John' } asDictionary
```

Dictionary pairs are composed with AND semantics. Selecting the elements having `John` as name AND whose `orders` value is greater than 10 can be written like this:

```
{
  'name' -> 'John'.
  'orders' -> { '$gt' : 10 } asDictionary
} asDictionary
```

To construct the ”greater than” statement, a new dictionary needs to be created that uses the MongoDB `$gt` query selector to express the greater than relation. For the list of available query selectors we refer to the MongoDB Query Selectors documentation[4].

Querying for an object by OID

If you know the ObjectId for a document, you can create an OID instance with this value and query for it.

```
{('_id' -> (OID value: 16r55CDD2B6E9A87A520F000001))} asDictionary.
```

Note that both of the following are equivalent:

```
OID value: 265550506989409955628365905 93. "dec"
OID value: 16r55CDD2B6E9A87A520F000001. "hex"
```

> **Note** If you have an instance which is in a root collection, then you can ask it for its `voyageId` and use that ObjectId in your query.

Using dot notation to access embedded documents

To access values embedded in documents with JSON queries, the dot notation is used. For example, the query representing rectangles whose origin have 42 as their `x` values can be expressed this way:

```
{
  'origin.x' -> {'$eq' : 42} asDictionary
} asDictionary
```

Expressing OR conditions in the query

To express an OR condition, a dictionary whose key is `'$or'` and whose values are the expression of the condition is needed. The following example shows how to select all objects whose name is `John` that have more than ten orders OR objects whose name is not `John` and has ten or less orders:

```
{ '$or' :
  {
    {
      'name' -> 'John'.
      'orders' -> { '$gt': 10 } asDictionary
```

[4] http://docs.mongodb.org/manual/reference/operator/query/#query-selectors

```
    } asDictionary.
    {
      'name' -> { '$ne': 'John'} asDictionary.
      'orders' -> { '$lte': 10 } asDictionary
    } asDictionary.
  }.
} asDictionary.
```

Going beyond MongoQueries features

Using JSON queries allows to use features that are not present in MongoQueries, for example the use of regular expressions. Below is a query that searches for all documents with a `fullname.lastName` that starts with the letter `D`:

```
{
  'fullname.lastName' -> {
    '$regexp': '^D.*'.
    '$options': 'i'.
  } asDictionary.
} asDictionary.
```

The option `i` for a regular expression means case insensitivity. More options are described in the documentation of the $regex operator[5].

This example only briefly illustrates the power of JSON queries. Many more different queries can be constructed, and the complete list of operators and usages is in the MongoDB operator documentation[6]

3.20 Executing a Query

Voyage has a group of methods to perform searches. To illustrate the use of these methods we will use the stored Point example we have presented before. Note that all queries in this section can be written either as MongoQueries or as JSON queries, unless otherwise specified.

3.21 Basic Object Retrieval

The following methods provide basic object retrieval.

- **`selectAll`** Retrieves all documents in the corresponding database collection. For example, `Point selectAll` will return all Points.
- **`selectOne:`** Retrieves one document matching the query. This maps to a `detect:` method and takes as argument a query specification (either a MongoQuery or a JSON Query). For example, `Point selectOne: [:each | each x = 42]` or alternatively `Point selectOne: { 'x' -> 42 } asDictionary`.
- **`selectMany:`** Retrieves all the documents matching the query. This maps to a `select:` method and takes as argument a query specification, like above.

3.22 Limiting Object Retrieval and Sorting

The methods that query the database look similar to their equivalent in the Collection hierarchy. However unlike regular collections which can operate fully on memory, often Voyage collection queries need to be customized in order to optimize memory consumption and/or access speed. This is because there can be literally millions of documents in each collection, surpassing

[5] http://docs.mongodb.org/manual/reference/operator/query/regex/#op._S_regex
[6] http://docs.mongodb.org/manual/reference/operator

the memory limit of Pharo, and also the database searches have a much higher performance than the equivalent code in Pharo.

The first refinement to the queries consist in limiting the amount of results that are returned. Of the collection of all the documents that match, a subset is returned that starts at the index that is given as argument. This can be used to only retrieve the first N matches to a query, or go over the query results in smaller blocks, as will be shown next in the simple paginator example.

- `selectMany:limit:` Retrieves a collection of objects from the database that match the query, up to the given limit. An example of this is `Point selectMany: [:each | each x = 42] limit: 10`
- `selectMany:limit:offset:` Retrieves a collection of objects from the database that match the query. The first object retrieved will be at the `offset` position plus one of the results of the query, and up to `limit` objects will be returned. For example, if the above example matched 25 points, the last 15 points will be returned by the query `Point selectMany: [:each | each x = 42] limit: 20 offset: 10` (any `limit` argument greater than 15 will do for this example).

The second customization that can be performed is to sort the results. To use this, the class `VOOrder` provides constants to specify `ascending` or `descending` sort order.

- `selectAllSortBy:` Retrieves all documents, sorted by the specification in the argument, which needs to be a JSON query. For example, `Point selectAllSortBy: { #x -> VOOrder ascending} asDictionary` returns the points in ascending x order.
- `selectMany:sortBy:` Retrieves all the documents matching the query and sorts them. For example to return the points where `x` is 42, in descending `y` order: `Point selectMany: { 'x' -> 42 } asDictionary sortBy: { #y -> VOOrder descending } asDictionary`.
- `selectMany:sortBy:limit:offset:` Provides for specifying a limit and offset to the above query.

3.23 A Simple Paginator Example

Often you want to display just a range of objects that belong to the collection, e.g. the first 25, or from 25 to 50, and so on. Here we present a simple paginator that implements this behavior, using the `selectMany:limit:offset:` method.

First we create a class named `Paginator`. To instantiate it, a Voyage root (`aClass`) and a query (`aCondition`) need to be given.

```
Object subclass: #Paginator
  instanceVariableNames: 'collectionClass where pageCount'
  classVariableNames: ''
  package: 'DemoPaginator'

Paginator class>>on: aClass where: aCondition
  ^ self basicNew
    initializeOn: aClass where: aCondition

Paginator>>initializeOn: aClass where: aCondition
  self initialize.
  collectionClass := aClass.
  where := aCondition
```

Then we define the arithmetic to get the number of pages for a page size and a given number of entities.

```
Paginator>>pageSize
  ^ 25
```

```
Paginator>>pageCount
  ^ pageCount ifNil: [ pageCount := self calculatePageCount ]

Paginator>>calculatePageCount
  | count pages |
  count := self collectionClass count: self where.
  pages := count / self pageSize.
  count \\ self pageSize > 0
    ifTrue: [ pages := pages + 1].
  ^ count
```

The query that retrieves only the elements for a given page is then implemented as follows:

```
Paginator>>page: aNumber
  ^ self collectionClass
    selectMany: self where
    limit: self pageSize
    offset: (aNumber - 1) * self pageSize
```

3.24 Creating and Removing Indexes

There are a number of useful features in MongoDB that are not present in Voyage but still can be performed from within Pharo, the most important one being the management of indexes.

3.25 Creating Indexes by using OSProcess

It is not yet possible to create and remove indexes from Voyage, but this can nonetheless be done by using OSProcess.

For example, assume there is a database named `myDB` with a collection named `Trips`. The trips have an embedded collection with receipts. The receipts have an attribute named `description`. The following creates an index on `description`:

```
OSProcess command:
  '/{pathToMongoDB}/MongoDB/bin/mongo --eval ',
  '"db.getSiblingDB(''myDB'').Trips.',
  'createIndex({''receipts.description'':1})"'
```

Removing all indexes on the Trips collection can be done as follows:

```
OSProcess command:
  '/{pathToMongoDB}/MongoDB/bin/mongo --eval ',
  '"db.getSiblingDB(''myDB'').Trips.dropIndexes()"'
```

3.26 Verifying the use of an Index

To ensure that a query indeed uses the index, `".explain()"` can be used in the mongo console. For example, if we add the index on `description` as above, run a query and add `.explain()` we see, that only a subset of documents were scanned.

```
> db.Trips.find({"receipts.description":"a"})
              .explain("executionStats")
{
  "cursor" : "BtreeCursor receipts.receiptDescription_1",
  "isMultiKey" : true,
  "n" : 2,
  "nscannedObjects" : 2,
  "nscanned" : 2,
  "nscannedObjectsAllPlans" : 2,
```

```
  "nscannedAllPlans" : 2,

  [...]
}
```

After removing the index, all documents are scanned (in this example there are 246):

```
> db.Trips.find({"receipts.description":"a"}
            ..explain("executionStats")
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 246,
  "nscanned" : 246,
  "nscannedObjectsAllPlans" : 246,
  "nscannedAllPlans" : 246,

  [...]
}
```

3.27 Conclusion

In this chapter we presented Voyage, a persistence programming framework. The strength of Voyage lies in the presence of the object-document mapper and MongoDB back-end. We have shown how to store objects in, and remove object from the database, and how to optimise the storage format. This was followed by a discussion of querying the database; showing the two ways in which queries can be constructed and detailing how queries are ran. We ended this chapter by presenting how we can construct indexes in MongoDB databases, even though Voyage does not provide direct support for it.

CHAPTER 4

Tips and Tricks

This chapter contains some tips and tricks that people collected over the years. It was written by Sabina Manaa.

4.1 How to query for an object by id?

If you know the _id value, you initialize an OID with this and query for it.

```
Person selectOne: {('_id' -> (OID value: 16r55CDD2B6E9A87A520F000001))} asDictionary.
```

Note that both are equivalent:

```
OID value: 265550506989409955628365905 93. "dec"
OID value: 16r55CDD2B6E9A87A520F000001. "hex"
```

Or you have an instance (in this example of `Person`) which is in a root collection, then you ask it for its voyageId and use it in your query. The following assumes that you have a `Trips` root collection and a `Persons` root collection. The trip has an embedded `receipts` collection. Receipts have a `description`. The query asks for all trips of the given person with at least one receipt with the description `aString`.

```
 Trip
    selectMany:
      {('receipts.description' -> aString).
      ('person._id' -> aPerson voyageId)} asDictionary
```

4.2 Not yet supported mongo commands

Indexes

It is not yet possible to create and remove indexes from voyage, but you can use OSProcess.

Assume you have a database named `myDB` with a collection named `Trips`. The trips have an embedded collection with receipts. The receipts have an attribute named `description`. Then you can create an index on description with

```
OSProcess command:
'/{pathToMongoDB}/MongoDB/bin/mongo --eval
    "db.getSiblingDB(''myDB'').Trips.createIndex({''receipts.description'':1})"'
```

Remove all indexes on the Trips collection with:

```
OSProcess command:
'/{pathToMongoDB}/MongoDB/bin/mongo --eval
    "db.getSiblingDB(''myDB'').Trips.dropIndexes()"'
```

Backup

It is not yet possible to create backup from voyage, so use

```
OSProcess command:
 '/{pathToMongoDB}/MongoDB/bin/mongodump  --out {BackupPath}'
```

Please see the mongo documentation for mongo commands, especially the `--eval` command.

4.3 Useful mongo commands

Use ".explain()" in the mongo console to ensure that your query indeed uses the index.

Example:

Create an index on an embedded attribute (`description`):

```
> db.Trips.createIndex({"receipts.description":1})
```

Query for it and call explain. We see, that only 2 documents were scanned. > db.Trips.find({"receipts.description":"a"}).explain("executionStats") { "cursor" : "BtreeCursor receipts.receiptDescription_1", "isMultiKey" : true, "n" : 2, "nscannedObjects" : 2, "nscanned" : 2, "nscannedObjectsAllPlans" : 2, "nscannedAllPlans" : 2, "scanAndOrder" : false, "indexOnly" : false, "nYields" : 0, "nChunkSkips" : 0, "millis" : 0, "indexBounds" : { "receipts.receiptDescription" : [["a", "a"]] }, "allPlans" : [{ "cursor" : "BtreeCursor receipts.receiptDescription_1", "n" : 2, "nscannedObjects" : 2, "nscanned" : 2, "indexBounds" : { "receipts.receiptDescription" : [["a", "a"]] } }], "server" : "MacBook-Pro-Sabine.local:27017" }]]]

Now, remove the index

```
> db.Trips.dropIndexes()
{
  "nIndexesWas" : 2,
  "msg" : "non-_id indexes dropped for collection",
  "ok" : 1
}
```

Query again, all documents were scanned.

```
> db.Trips.find({"receipts.receiptDescription":"a"}).explain("executionStats")
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 246,
  "nscanned" : 246,
  "nscannedObjectsAllPlans" : 246,
  "nscannedAllPlans" : 246,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 1,
  "indexBounds" : {

  },
  "allPlans" : [
    {
```

```
      "cursor" : "BasicCursor",
      "n" : 2,
      "nscannedObjects" : 246,
      "nscanned" : 246,
      "indexBounds" : {

      }
    }
  ],
  "server" : "MacBook-Pro-Sabine.local:27017"
}
```

4.4 Storing instances of Date in Mongo

A known issue of mongo is Mongo that it does not difference between `Date` and `DateAndTime`, so even if you commit a `Date`, you will get back a `DateAndTime`. You have to transform it back to `Date` manually when materializing the object.

4.5 Database design

Often you objects do not form a simple tree but a graph with cycles. For example you could have persons which are pointing to their trips and each trip knows about its person (`Person <->>Trip`). If you create a root Collection with Persons and a Root collection with Trips, you avoid endless loops to be generated (see chapter 1.2).

This is an example for a `trip` pointing to a `person` which is in another root collection and another root collection, `paymentMethod`. Note that the receipt also points back to the trip, which does not create a loop.

```
Trip
{
 "_id" : ObjectId("55cf2bc73c9b0fe702000008"),
"#version" : 876079653,
"person" : {
  "#collection" : "Persons",
  "_id" : ObjectId("55cf2bbb3c9b0fe702000007") },
"receipts" : [
  { "currency" : "EUR",
  "date" : { "#instanceOf" : "ZTimestamp", "jdn" : 2457249, "secs" : 0 },
  "exchangeRate" : 1,
  "paymentMethod" : {
    "#collection" : "PaymentMethods",
    "_id" : ObjectId("55cf2bbb3c9b0fe702000003") },
  "receiptDescription" : "Taxi zum Hotel",
  "receiptNumber" : 1,
  "trip" : {
    "#collection" : "Trips",
    "_id" : ObjectId("55cf2bc73c9b0fe702000008") } } ],
  "startPlace" : "Österreich",
  "tripName" : "asdf",
  "tripNumber" : 1 }
```

The corresponding `person` points to all its `trips` and to its `company`.

```
{ "#version" : 714221829,
"_id" : ObjectId("55cf2bbb3c9b0fe702000007"),
"bankName" : "",
 "company" : {
"#collection" : "Companies",
```

```
"_id" : ObjectId("55cf2bbb3c9b0fe702000002") },
"email" : "bb@spesenfuchs.de",
"firstName" : "Berta",
"lastName" : "Block",
"roles" : [  "user" ],
"tableOfAccounts" : "SKR03",
"translator" : "German",
"trips" : [
{
"#collection" : "Trips",
"_id" : ObjectId("55cf2bc73c9b0fe702000008") } ] }
```

If your domain has strictly delimited areas, e.g. clients, you could think about creating one repository per area (client).

4.6 Retrieving data

One question is if it possible to retrieve data from Mongo collection even if the database was not created via Voyage. Yes it is possible. Here is the solution.

First we create a class `MyClass` with two class side methods:

```
MyClass class >> isVoyageRoot
  ^ true
```

```
MyClass class >> descriptionContainer
    <voyageContainer>
    ^ VOContainer new
        collectionName: 'myCollection';
        yourself
```

Also, to properly read the data one should add instance variables depending on what is in the database.

For example if we have the following information stored in the database:

```
{ "_id" : ObjectId("5900a0175bc65a2b7973b48a"), "item" : "canvas", "qty" : 100, "tags" :
    [ "cotton" ] }
```

In this case `MyClass` should have instanceVariables: `item`, `qty` and `tags` and accessors. Then we define the following description on the class side

```
MyClass class >> mongoItem
  <mongoDescription>
  ^ VOToOneDescription new
    attributeName: 'item';
    kind: String;
    yourself
```

```
MyClass class >> mongoQty
  <mongoDescription>
  ^ VOToOneDescription new
    attributeName: 'qty';
    kind: Integer;
    yourself
```

```
MyClass class >> mongoTags
  <mongoDescription>
  ^ VOToOneDescription new
    attributeName: 'tags';
    kind: OrderedCollection;
    yourself
```

After that one can connect to database and get the information.

```
| repository |
repository := VOMongoRepository database: 'databaseName'.
repository selectAll: MyClass
```