CHAPTER 1

TinyBlog: Data Persitency using Voyage and Mongo

1.1 Previous Week Solution

You can load the solution of the previous week using the following snippet:

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #week3solution;
    configuration: 'TinyBlog';
    load
```

Then, test it with:

```
TBTeapotWebApp start
```

Pay attention that you should not have your own web application running on the same port. If this is the case stop it or run the solution on a different port. You should also create posts in the blog to be able to see data.

```
TBBlog reset ; createDemoPosts
```

1.2 Saving in an External Database with Voyage

Until now we used model objects stored in memory and it works well because saving the Pharo image also save these objects. Nevertheless, it would be better to save these model data into an external database. Voyage provides a mean to store objects into a Mongo database. In fact Voyage is a framework that offers the same API to document-based databases such as Mongo or Un-

QLite. It is what we will discover in this exercise. First, we will use Voyage and its capacity to simulate an external database. This is really useful during development. Then, we will install a Mongo database and access it through Voyage. This second step will have a really little impact on our code.

1.3 Configure Voyage to Save TBBlog Objects

By defining the class method `isVoyageRoot`, we declare that objects of this class must be saved into the database as root objects.

```
TBBlog class >> isVoyageRoot
   "Indicates that instances of this class are top level document in
    noSQL databases"
   ^ true
```

We should establish connection to real database or work in memory. We will start to work in memory by using this expression:

```
VOMemoryRepository new enableSingleton.
```

The `enableSingleton` message indicates to Voyage that we will use only one database. This will free us to specify the database each time.

We create and initialize the database in memory in a class-side method named `initializeVoyageOnMemoryDB`.

```
TBBlog class >> initializeVoyageOnMemoryDB
   VOMemoryRepository new enableSingleton
```

We do not need to save the database into an instance variable because we will have only one database accessible through Voyage configured in singleton mode.

We now define the `reset` and `initialize` class methods to ensure that the databse is initialized when we load TinyBlog's code.

The `reset` method re-initializes the database.

```
TBBlog class >> reset
   self initializeVoyageOnMemoryDB

TBBlog class >> initialize
   self reset
```

The class-side `current` method is more tricky. Before using Voyage, we implemented a simple singleton pattern. However, it does not work anymore because imagine that we saved our blog and then the server stop by accident. When we would reload a new version of the code, it would re-initializes the connection and create a new instance. It would then be possible to end up with a different instance than the saved one.

So we change the implementation of the `current` method to make a database request and retrieve saved objects. Since we only save one blog object, it only consists in doing: `self selectOne: [ :each | true ]` or `self selectAll anyOne`. If the database contains no instance, we create a new one and save it.

```
TBBlog class >> current
   ^ self selectAll
         ifNotEmpty: [ :x | x anyOne ]
         ifEmpty: [ self new save ]
```

We can also remove the class instance variable named `uniqueInstance` that we previously used to store our singleton object.

1.4 Saving a Blog

Each time we modify a blog object, we must propagate changes into the database. For example, we modify the `writeBlogPost:` method to save the blog when we add a new post.

```
TBBlog >> writeBlogPost: aPost
   "Write the blog post in database"
   self allBlogPosts add: aPost.
   self save
```

We also save the blog when removing (`remove` method) a post from a blog.

```
TBBlog >> removeAllPosts
   posts := OrderedCollection new.
   self save
```

1.5 Querying the Database

The database is currently in memory and we can access to the blog object using the `current` class-side method of the `TBBlog` class. It is enough to show the API of Voyage since it will be the same to access a real Mongo database.

You can create posts:

```
TBBlog createDemoPosts.
```

You can count the number of blog saved. `count` is part of the Voyage API. In this example, we get the result 1 because the blog is implemented as a Singleton.

```
TBBlog count
>1
```

Similarly, you can retrieve all saved root objects of one kind.

```
TBBlog selectAll
```

You can also remove one root objet.

```
TBBlog current remove
```

You can discover more about the Voyage API by looking at:

- the `Class` class,
- the `VORepository` class which is the root of the hierachy of all databases either in memory or external.

Those queries will be more relevant when there will more objects but they would be similar.

1.6 If we would save Posts

This section should not be implemented. It is only described as an example. More information about Voyage can be found in the Entreprise Pharo book http://books.pharo.org. We want to illustrate that declaring a class as Voyage root has an influence on how an instance of this class is saved and reloaded since it is not part anymore of the object that refers to it. In the database this object is not nested anymore inside the document representing the object referencing it but it is saved as an autonomous document.

We could define that a post can be saved independently from a blog. However, if we represent post comments, we would not define them as root objects too because manipulating a comment outside of its context (a post) does not make sense.

When a post is not declared as voyage root, you are not sure that this post is unique in the sense that you have only one object representing it when loaded from the base. Indeed, when loading (and this can be different from the situation before the saving) a post is nested inside the document representing the blog and as such it cannot be shared between two blog instances therefore it may be duplicated.

Post as Root = Uniqueness

Declaring posts as root objects would imply that saved blogs have a reference to a `TBPost` object. Otherwise, a post is included into its blog. When posts are not root objects, they are not guaranteed to be unique after loading from the database. Indeed, after loading each blog object will have its own posts objects even if some posts were shared before saving. Shared objects before saving will be duplicated for each root objects after loading.

If you want to share posts and make them unique between multiple blogs, therefore, the `TBPost` class must be declared as a root in the database. In this

case, posts are saved as autonomous entities and instances of `TBBlog` will reference posts entities instead of embedding them. The consequence is that a post is unique and can be shared via reference from a blog. To achieve this, we **would** define the following methods:

```
TBPost class >> isVoyageRoot
   "Indicates that instances of this class are top level document in
    noSQL databases"
   ^ true
```

During the addition of a post to a blog, it would be important to save both the blog and the new post.

```
TBBlog >> writeBlogPost: aPost
   "Write the blog post in database"
   posts add: aPost.
   aPost save.
   self save
```

```
TBBlog >> removeAllPosts
   posts do: [ :each | each remove ].
   posts := OrderedCollection new.
   self save.
```

In the `removeAllPosts` method, we first remove all posts, then update the collection and finally save the blog.

1.7 Test with an External Mongo Database

By using Voyage, we can easily save our model objects into a Mongo database. This section explains how to proceed and the few modifications to make into our code.

1.8 Installing Mongo

Regardless of your operating system (Linux, Mac OSX ou Windows), you can install a local Mongo server on your machine. This is useful to test your application without requiring an internet connection. We will not describe here the process to install Mongo on various systems, please read the Mongo documentation.

> **Note** The running Mongo server must not use authentication because the new SCRAM mechanism used by Mongo 3.0 is currently not supported by Voyage.

Once you successfully installed Mongo locally, you can access it from Pharo. You should create a database named 'tinyblog'. It will be used to save our objects.

1.9 Connecting a Local Server

We define the method named `initializeLocalhostMongoDB` to establish the connection to a Mongo server running locally (localhost) on the default Mongo port.

```
TBBlog class >> initializeLocalhostMongoDB
   | repository |
   repository := VOMongoRepository database: 'tinyblog'.
   repository enableSingleton.
```

Now we make sure that resetting the class will set a connection to the database.

```
TBBlog class >> reset
   self initializeLocalhostMongoDB
```

Now you can use your application. For example add new posts. They will be saved directly in the Mongo database.

1.10 In case of trouble

If you need to re-initalize completely an external database, you can use the `dropDatabase` method.

```
(VOMongoRepository
   host: 'localhost'
   database: 'tinyblog') dropDatabase
```

1.11 Points of Attention: Changing TBBlog Definition

When you use an external Mongo database instead of a memory one, each time you add new root objects or modify the definition of some root objects, it is important to reset the cache maintained by Voyage. It can be done using:

```
   VORepository current reset
```

1.12 Conclusion

Voyage proposes a nice API to manage transparently storage of objects either into memory or in document database. Your application can now be saved into a database and we are now ready for its web user interface.