TinyBlog: Develop your First Web App with Pharo

Olivier Auverlot, Stéphane Ducasse and Luc Fabresse

April 28, 2019
## Contents

### Illustrations

1 About this book .......................... 1
   1.1 Structure .................................. 1
   1.2 Pharo Installation ......................... 2
   1.3 Naming Rules .............................. 3
   1.4 Resources .................................. 3

### Core Tutorial

2 TinyBlog Application: Core model .......... 7
   2.1 TBPost Class .................................. 7
   2.2 Post Visibility ............................... 8
   2.3 Initialization ................................. 9
   2.4 Posts Creation Methods ....................... 9
   2.5 Creating a Post .............................. 10
   2.6 Adding Some Unit Tests ...................... 10
   2.7 Post Queries .................................. 11
   2.8 Conclusion .................................... 12

3 TinyBlog: Extending and Testing the Model .... 13
   3.1 TBBlog class .................................. 13
   3.2 Only One Blog Object ......................... 14
   3.3 Testing the Model .............................. 14
   3.4 A First Test .................................. 15
   3.5 Increasing Test Coverage ...................... 16
   3.6 Other Functionalities .......................... 16
   3.7 Testing data .................................. 18
   3.8 Possible Extensions ............................ 19
   3.9 Conclusion .................................... 19
II Optional Elements

11 Loading Chapter Code 97
11.1 Chapter 3: Extending and Testing the Model .............................. 97
11.2 Chapter 4: Data Persistence using Voyage and Mongo .................. 97
11.3 Chapter 5: First Steps with Seaside ........................................ 98
11.4 Chapitre 6: Web Components for TinyBlog .............................. 98
11.5 Chapitre 7: Managing Categories ............................................. 98
11.6 Chapitre 8: Authentication and Session .................................... 99
11.7 Chapitre 9: Administration Web Interface and Automatic Form Generation 99
11.8 Chapitre 10: Deploying TinyBlog ............................................ 99

12 Save your code 101
12.1 Up to Pharo 6.0 ............................................................ 101
12.2 With Pharo 70 ............................................................... 102
1-1 The TinyBlog application. ........................................... 2
2-1 TBPost: a really basic class mostly handling data. ............... 7
2-2 Inspector on a TBPost instance. .................................. 10
3-1 TBBlog: A simple class containing posts. ........................ 13
5-1 Starting the Seaside server. ....................................... 29
5-2 Running Seaside. .................................................... 30
5-3 Browsing the Seaside Bootstrap Library. .......................... 31
5-4 A Bootstrap element and its code. ................................ 32
5-5 TinyBlog is a registered Seaside application. ........................ 33
5-6 A first Seaside web page. ......................................... 33
5-7 Main components of TinyBlog (public view). ........................ 34
5-8 Architecture of TinyBlog. ........................................... 35
6-1 Component Architecture of the Public View (opposed to the Administration View). ................................................................. 37
6-2 Visual Components of TinyBlog. .................................... 38
6-3 ApplicationRootComponent temporarily uses a ScreenComponent that contains a HeaderComponent. ..................................................... 39
6-4 First visual rendering of TBScreenComponent. ....................... 40
6-5 TinyBlog with a Bootstrap header. ................................... 42
6-6 The ApplicationRootComponent uses PostsListComponent. .......... 43
6-7 TinyBlog displaying a basic posts list. ............................... 44
6-8 Using PostComponents to displays each Posts. ..................... 44
6-9 TinyBlog with a List of Posts. ....................................... 46
7-1 L’architecture des composants de la partie publique with categories. ..... 49
7-2 Categories and Posts. ................................................. 53
7-3 Post list with a better layout. ....................................... 54
7-4 Final TinyBlog Public UI. ............................................. 56
<table>
<thead>
<tr>
<th>Illustrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-1</td>
</tr>
<tr>
<td>8-2</td>
</tr>
<tr>
<td>8-3</td>
</tr>
<tr>
<td>8-4</td>
</tr>
<tr>
<td>8-5</td>
</tr>
<tr>
<td>8-6</td>
</tr>
<tr>
<td>8-7</td>
</tr>
<tr>
<td>9-1</td>
</tr>
<tr>
<td>9-2</td>
</tr>
<tr>
<td>9-3</td>
</tr>
<tr>
<td>9-4</td>
</tr>
<tr>
<td>9-5</td>
</tr>
<tr>
<td>9-6</td>
</tr>
<tr>
<td>9-7</td>
</tr>
<tr>
<td>10-1</td>
</tr>
<tr>
<td>10-2</td>
</tr>
</tbody>
</table>
In this book, we will guide you to develop a mini project: a small web application, named TinyBlog, that manages a blog system (see its final state in Figure 1-1). The idea is that a visitor of the web site can read the posts and that the post author can connect to the web site as admin to manage its posts (add, remove and modify existing ones).

TinyBlog is a small pedagogical application that will show you how to define and deploy a web application using Pharo / Seaside / Mongo and frameworks available in Pharo such as NeoJSON.

Our goal is that you will be able to reuse and adapt such an infrastructure to create your own web applications.

1.1 **Structure**

In the first part called "Core Tutorial", you will develop and deploy, TinyBlog, an application and its administration using Pharo, the Seaside application web server framework as well as some other frameworks such as Voyage and Magritte. Deployment with Mongo DB is optional but it allows you to see that Voyage is an elegant facade to persist your data within Mongo.

In the second part and optional part, we will show you some optional aspects such as data export, use of Mustache or how to expose your application using a REST API.

Presented solutions are sometimes not the best. This is done that way to offer you a room for improvement. Our goal is not to be exhaustive. We present one way to develop TinyBlog nevertheless we invite the reader to read further references such as books or tutorials on Pharo to deepen his expertise and enhance his application.
Finally, to help you to get over possible errors and avoid to get stuck, the last chapter describes how to load the code described in each chapter.

1.2 Pharo Installation

In this tutorial, we suppose that you are using Pharo 6.1 with an image in which many frameworks and web libraries have been loaded: Seaside (component-based web application server), Magritte (an automatic generation report system based on descriptions), Bootstrap (a library to visually tune web applications), Voyage (a framework to save your objects in document databases) and some others.

To develop your application and follow this tutorial, we suggest to use the image available at the following URL: http://mooc.pharo.org/image/PharoWeb-60.zip since it contains all the mandatory packages.

You can rebuild this web Pharo image in Pharo 6.1 (http://pharo.org/download) with the following script:
1.3 Naming Rules

In the following, we prefix all the class names TB (for TinyBlog). You may:

- either choose another prefix (by example TBM) to be able to load the solution side by side to your own. This way you will be able to compare the two solutions,
- either choose the same prefix to fusion the proposed solutions in your code. The merge tool will help you see the differences and learn from the changes. This solution may be more complex if you implement your own extra functionalities.

1.4 Resources

Pharo has many strong pedagogial resources as well as a super friendly community of users. Here is a list of resources:


- http://mooc.pharo.org proposes an excellent Mooc with more that 90 videos explaining syntactically points as well as object programming key concepts.

- http://discord.gg/Sj2rhxn is a discord channel where many Pharoers exchange information and help each other.
Part I

Core Tutorial
In this chapter, we start to develop a part of the domain model of TinyBlog. The model is particularly simple: it starts with a post. In the next chapter we will add a blog containing a list of posts.

2.1 TBPost Class

We start with the post representation. It is super simple as shown by Figure 2-1. It is defined by the class TBPost:

```object
Object subclass: #TBPost
  instanceVariableNames: 'title text date category visible'
  classVariableNames: ''
  package: 'TinyBlog'
```

A blog post is described by 5 instance variables.

<table>
<thead>
<tr>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
</tr>
<tr>
<td>date</td>
</tr>
<tr>
<td>title</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>category</td>
</tr>
<tr>
<td>isVisible</td>
</tr>
<tr>
<td>isUnclassified</td>
</tr>
</tbody>
</table>

Figure 2-1 TBPost: a really basic class mostly handling data.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Signification</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>post title</td>
</tr>
<tr>
<td>text</td>
<td>post text</td>
</tr>
<tr>
<td>date</td>
<td>date of writing</td>
</tr>
<tr>
<td>category</td>
<td>name of the category of the post</td>
</tr>
<tr>
<td>visible</td>
<td>is the post publicly visible or not?</td>
</tr>
</tbody>
</table>

All of these variables have corresponding accessor methods in the 'accessing' protocol. You can use a refactoring to automatically create all the following methods:

```small
TBPost >> title
^ title
```

```small
TBPost >> title: aString
  title := aString
```

```small
TBPost >> text
^ text
```

```small
TBPost >> text: aString
  text := aString
```

```small
TBPost >> date
^ date
```

```small
TBPost >> date: aDate
  date := aDate
```

```small
TBPost >> visible
^ visible
```

```small
TBPost >> visible: aBoolean
  visible := aBoolean
```

```small
TBPost >> category
^ category
```

```small
TBPost >> category: anObject
  category := anObject
```

### 2.2 Post Visibility

We should add methods to make a post visible or not and also test if it is visible. Those methods are defined in the 'action' protocol.

```small
TBPost >> beVisible
  self visible: true
```

```small
TBPost >> notVisible
  self visible: false
```
2.3 Initialization

The initialize method ('initialization' protocol) sets the date to the current day and the visibility to false: the user must explicitly make a post visible. This allows him to write drafts and only publish a post when the post is finished. By default, a post belongs to the 'Unclassified' category that we define at the class level. This category name is defined on class-side by the unclassifiedTag method.

```smalltalk
TBPost class >> unclassifiedTag
  ^ 'Unclassified'
```

Pay attention the method `unclassifiedTag` should be defined on the class-side of the class `TBPost` (click on the `class` button to define it). The other methods are defined on the instance-side: it means that they will be applied to `TBBlog` instances.

```smalltalk
TBPost >> initialize
  super initialize.
  self category: TBPost unclassifiedTag.
  self date: Date today.
  self notVisible
```

In the solution above, it would be better that the initialize method does not hard code the reference to the `TBPost` class. Propose a solution. The sequence 3 of the week 6 of the Mooc can help you to understand why it is better to avoid hardcoding class references (See http://mooc.pharo.org).

2.4 Posts Creation Methods

On class-side, we add class methods (i.e. methods execute on class) to ease posts creation for blogs - usually such kind of methods are grouped in the protocol 'instance creation'.

We define two methods.

```smalltalk
TBPost class >> title: aTitle text: aText
  ^ self new
  title: aTitle;
  text: aText;
  yourself

TBPost class >> title: aTitle text: aText category: aCategory
  ^(self title: aTitle text: aText)
  category: aCategory;
  yourself
```
2.5 Creating a Post

Let us create posts to check a bit the created objects. Using the Playground tools execute the following expression:

```plaintext
TBPost
  title: 'Welcome in TinyBlog'
  text: 'TinyBlog is a small blog engine made with Pharo.'
  category: 'TinyBlog'
```

When you inspect the code above (right click and "Inspect it"), you will obtain an inspector on the newly created object as shown in Figure 2-2.

2.6 Adding Some Unit Tests

Manually looking at objects is not a way to systematically verifying that such objects follow some expected invariant. Even though the model is quite simple we can define some tests. In Test Driven Development mode we write test first. Here we preferred to let you define a little class to familiarize with the IDE. Let us fix this!

We define the class `TBPostTest` (as subclass of the class `TestCase`).
Let us define a two tests.

```smalltalk
TBPostTest >> testWithoutCategoryIsUnclassified

| post |
post := TBPost
   title: 'Welcome to TinyBlog'
   text: 'TinyBlog is a small blog engine made with Pharo.'.
self assert: post title equals: 'Welcome to TinyBlog'.
self assert: post category = TBPost unclassifiedTag.

TBPostTest >> testPostIsCreatedCorrectly

| post |
post := TBPost
   title: 'Welcome to TinyBlog'
   text: 'TinyBlog is a small blog engine made with Pharo.'.
   category: 'TinyBlog'.
self assert: post title equals: 'Welcome to TinyBlog'.
self assert: post text equals: 'TinyBlog is a small blog engine made with Pharo.'.
```

Your tests should pass.

## 2.7 Post Queries

In the protocol 'testing', define the following methods that checks whether a post is visible, and whether it is classified or not.

```smalltalk
TBPost >> isVisible
^ self visible

TBPost >> isUnclassified
^ self category = TBPost unclassifiedTag
```

It is not really good to harcode a reference to the class TBPost in a method body. Propose a solution.

In addition, let us take the time to update our test to take advantage of the new behavior.

```smalltalk
TBPostTest >> testWithoutCategoryIsUnclassified

| post |
post := TBPost
   title: 'Welcome to TinyBlog'
   text: 'TinyBlog is a small blog engine made with Pharo.'.
```
self assert: post title equals: 'Welcome to TinyBlog'.
self assert: post isUnclassified.
self deny: post isVisible

2.8 **Conclusion**

We develop a first part of the model (the class TBPost) and some tests. We strongly suggest writing some other unit tests to make sure that your model fully work.
TinyBlog: Extending and Testing the Model

In this chapter we extend the model and add more tests. Note that when you will get fluent in Pharo, you will tend to write first your tests and then execute tests to code in the debugger. We did not do it because coding in the debugger requires more explanation. You can see such a practice in the Mooc video entitled Coding a Counter in the Debugger (See http://mooc.pharo.org) and read the book Learning Object-Oriented Programming, Design with TDD in Pharo (http://books.pharo.org).

Before starting, use back the code of the previous chapter or use the information of Chapter ??.

3.1 TBBlog class

We develop the class TBBlog that contains posts (as shown by Figure 3-1). We define some unit tests.

Here is its definition:

![Figure 3-1](image-url)
Object subclass: #TBBlog  
\begin{verbatim}
  instanceVariableNames: 'posts'
  classVariableNames: ''
  package: 'TinyBlog'
\end{verbatim}

We initialize the `posts` instance variable to an empty collection.

```smalltalk
TBBlog >> initialize
  super initialize.
  posts := OrderedCollection new
```

## 3.2 Only One Blog Object

In the rest of this project, we assume that we will manage only one blog. Later, you may add the possibility to manage multiple blogs such as one per user of the TinyBlog application. Currently, we use a Singleton design pattern on the `TBBlog` class. However pay attention since this pattern introduces a kind of global variable in your application and brings less modularity to your system. Therefore avoid to make explicit references to the singleton, better use an instance variable whose value first refers to the singleton so that later you can pass another object without being forced to rewrite everything. Do not generalize what we are doing for this class.

Since all the management of a singleton is a class behavior, we define such methods at the class level of `TBBlog`. We define an instance variable at the class level:

```smalltalk
TBBlog class
  instanceVariableNames: 'uniqueInstance'
```

Then we define two methods to manage the singleton.

```smalltalk
TBBlog class >> reset
  uniqueInstance := nil

TBBlog class >> current
  "answer the instance of the TBRepository"
  ^ uniqueInstance ifNil: [ uniqueInstance := self new ]
```

We redefine the class method `initialize` so that when the class is loaded in memory the singleton got reset.

```smalltalk
TBBlog class >> initialize
  self reset
```

## 3.3 Testing the Model

We now adopt a Test-Driven Development approach i.e., we will write a unit test first and then develop the functionality until the test is green. We will repeat this process for each functionality of the model.
We create unit tests in the TBBlogTest class that belongs to the TinyBlog-Tests tag. A tag is just a label to sort classes inside a package (See menu item 'Add Tag...'). We use a tag because using two packages will make this project more complex. However, while implementing a real application, it is recommended to have one (or multiple) separate test packages.

```smalltalk
TestCase subclass: #TBBlogTest
    instanceVariableNames: 'blog post first'
    classVariableNames: ''
    package: 'TinyBlog-Tests'
```

Before each test execution, the `setUp` method initializes the test context (also called test fixture). For example, it erases the blog content, adds one post and creates another temporary post that is not saved.

Pay attention since we will have to modify such behavior in the future else each time we will run the test we will destroy our data. This is an example of the kind of insidious behavior that a singleton introduces.

```smalltalk
TBBlogTest >> setUp
    blog := TBBlog current.
    blog removeAllPosts.

    first := TBPost title: 'A title' text: 'A text' category: 'First Category'.
    blog writeBlogPost: first.

    post := (TBPost title: 'Another title' text: 'Another text'
        category: 'Second Category') beVisible
```

As you may notice, we test different configurations. Posts do not belong to the same category, one is visible and the other is not visible.

At the end of each test, the `tearDown` method is executed and resets the blog.

```smalltalk
TBBlogTest >> tearDown
    TBBlog reset
```

Here we see one of the limits of using a Singleton. Indeed, if you deploy a blog and then execute the tests, you will lose all posts that have been created because we reset the Blog singleton. We will address this problem in the future.

We will now develop tests first and then implement all functionalities to make them green.

### A First Test

The first test adds a post in the blog and verifies that this post is effectively added.
If you try to execute it, you will notice that this test is not green (does not pass) because we did not define the methods `writeBlogPost:`, `removeAllPosts` and `size`. Let’s add them:

```small
TBBlog >> removeAllPosts
  posts := OrderedCollection new

TBBlog >> writeBlogPost: aPost
  "Add the blog post to the list of posts."
  posts add: aPost

TBBlog >> size
  ^ posts size
```

The previous test should now pass (i.e. be green).

### 3.5 Increasing Test Coverage

We should also add tests to cover all functionalities that we introduced.

```small
TBBlogTest >> testSize
  self assert: blog size equals: 1

TBBlogTest >> testRemoveAllBlogPosts
  blog removeAllPosts.
  self assert: blog size equals: 0
```

### 3.6 Other Functionalities

We follow the test-driven way of defining methods: First we define a test. Then we verify that this test is failing. Then we define the method under test and finally verify that the test passes.

#### All Posts

Let’s a test that fails:

```small
TBBlogTest >> testAllBlogPosts
  blog writeBlogPost: post.
  self assert: blog allBlogPosts size equals: 2
```

And the model code that makes it succeed:

```small
TBBlog >> allBlogPosts
  ^ posts
```

Your test should pass.
3.6 Other Functionalities

Visible Posts

We define a new unit test accessing visible blogs:

```smalltalk
TBBlogTest >> testAllVisibleBlogPosts
    blog writeBlogPost: post.
    self assert: blog allVisibleBlogPosts size equals: 1
```

We add the corresponding method:

```smalltalk
TBBlog >> allVisibleBlogPosts
    ^ posts select: [:p | p isVisible ]
```

Verify that the test passes.

All Posts of a Category

The following test verifies that we can access all the posts of a given category. Once defined, we should make sure that the test failed.

```smalltalk
TBBlogTest >> testAllBlogPostsFromCategory
    self assert: (blog allBlogPostsFromCategory: 'First Category') size equals: 1
```

Then we can define the functionality and make sure that our test passes.

```smalltalk
TBBlog >> allBlogPostsFromCategory: aCategory
    ^ posts select: [:p | p category = aCategory ]
```

Verify that the test passes.

All visible Posts of a Category

The following test verifies that we can access all the visible posts of a given category. Once defined, we should make sure that the test failed.

```smalltalk
TBBlogTest >> testAllVisibleBlogPostsFromCategory
    blog writeBlogPost: post.
    self assert: (blog allVisibleBlogPostsFromCategory: 'First Category') size equals: 0.
    self assert: (blog allVisibleBlogPostsFromCategory: 'Second Category') size equals: 1
```

Then we can define the functionality and make sure that our test passes.

```smalltalk
TBBlog >> allVisibleBlogPostsFromCategory: aCategory
    ^ posts select: [:p | p category = aCategory and: [ p isVisible ] ]
```

Verify that the test passes.
Check unclassified posts

The following test verifies that we do not have unclassified blogs in our test fixture.

```smalltalk
TBBlogTest >> testUnclassifiedBlogPosts
    self assert: (blog allBlogPosts select: [:p | p isUnclassified]) size equals: 0
```

Verify that the test passes.

Retrieve all categories

Again we define a new test and verify that it fails.

```smalltalk
TBBlogTest >> testAllCategories
   blog writeBlogPost: post.
   self assert: blog allCategories size equals: 2
```

We then add the new behavior.

```smalltalk
TBBlog >> allCategories
    ^ (self allBlogPosts collect: [:p | p category]) asSet
```

Verify that the test passes.

3.7 Testing data

To help you testing the application, you can add the following method that creates multiple posts.

```smalltalk
TBBlog class >> createDemoPosts
   "TBBlog createDemoPosts"
   self current
       writeBlogPost: ((TBPost title: 'Welcome in TinyBlog' text: 'TinyBlog is a small blog engine made with Pharo.' category: 'TinyBlog') visible: true);
       writeBlogPost: ((TBPost title: 'Report Pharo Sprint' text: 'Friday, June 12 there was a Pharo sprint / Moose dojo. It was a nice event with more than 15 motivated sprinters. With the help of candies, cakes and chocolate, huge work has been done' category: 'Pharo') visible: true);
       writeBlogPost: ((TBPost title: 'Brick on top of Bloc - Preview' text: 'We are happy to announce the first preview version of Brick, a new widget set created from scratch on top of Bloc. Brick is being developed primarily by Alex Syrel (together with Alain Plantec, Andrei Chis and myself), and the work is sponsored by ESUG.
           Brick is part of the Glamorous Toolkit effort and will provide the basis for the new versions of the development tools.' category: 'Pharo') visible: true);
```
3.8 Possible Extensions

Many extensions can be made such as: retrieve the list of categories that contains at least one visible post, delete a category and all posts that it contains, rename a category, move a post from one category to another, make (in)visible one category and all its content, etc. We encourage you to develop some of them.

3.9 Conclusion

You now have the full model of TinyBlog as well as some unit tests. You are now ready to implement more advanced functionality such as the database storage or a first Web front-end. Do not forget to save your code.
Until now we used model objects stored in memory and it works well because saving the Pharo image also saves these objects. Nevertheless, it would be better to save these objects (blog posts) into an external database. Pharo supports multiple object serializers such as Fuel (binary format) or STON (text format). These serializers are useful and powerful. Often with a single line of code we can save a full graph on objects as explained in the Enterprise Pharo book available at http://books.pharo.org.

In this chapter, we will use another possibility: saving data in a document database such as Mongo (https://www.mongodb.com) using the Voyage framework. Voyage provides a unified API to store and retrieve objects in various document-based databases such as Mongo or UnQLite. But first, we will use Voyage and its capacity to simulate an external database in memory. This is really useful during development. Then, you may install a local Mongo database and access it through Voyage. As you will see, this second step will have a really little impact on our code.

The last chapter explains how to load the code of previous chapters if needed.

### 4.1 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. It means that the database will contain as many documents as instances of this class.
Data Persistence using Voyage and Mongo

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

We should establish connection to real database or work in memory. Let’s 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

The reset class method re-initializes the database. The initialize class method ensures that the database is initialized when we load TinyBlog’s code. Do not forget to execute this expression TBBlog initialize to ensure that the database is initialized.

TBBlog class >> reset
self initializeVoyageOnMemoryDB

TBBlog class >> initialize
self reset

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

So we change the implementation of the current class 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.

TBBlog class
instanceVariableNames: ''
4.2 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.

```smalltalk
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.

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

4.3 Revising Unit Tests

We now save blogs in a database, either in memory or in an external Mongo server, through Voyage. We must be careful with unit tests that modify the database because they may corrupt production data. To circumvent this dangerous situation, a test should not modify the state of the system.

To solve this situation, before running a test we will keep a reference to the current blog and create a new context and restore it after test execution.

Let's add an instance variable `previousRepository` in the `TBBlogTest` class.

```smalltalk
TestCase subclass: #TBBlogTest
  instanceVariableNames: 'blog post first previousRepository'
  classVariableNames: '
  package: 'TinyBlog-Tests'
```

Then, we modify the `setUp` method to save the database before each test execution. We create a temporary database object that will be used by the test.

```smalltalk
TBBlogTest >> setUp
  previousRepository := VORespository current.
  VORespository setRepository: VOMemoryRepository new.
  blog := TBBlog current.
  first := TBPost title: 'A title' text: 'A text' category: 'First Category'.
  blog writeBlogPost: first.
  post := (TBPost title: 'Another title' text: 'Another text'
            category: 'Second Category') beVisible
```

In the `tearDown` method executed after each test, we restore the original database object.
4.4 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:

```small
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.

```small
TBBlog count > 1
```

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

```small
TBBlogselectAll
```

You can also remove a root object using the `remove` message.

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

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

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

4.5 If we would Save Posts [Discussion]

This section should not be implemented. It is only described as an example (More information about Voyage can be found in the Enterprise Pharo book http://books.pharo.org). We want to illustrate that declaring a class as a Voyage root has an influence on how an instance of this class is saved and reloaded.

So far, a post (an instance of TBPPost) is not declared as a Voyage root. Post objects are therefore saved as sub-parts into the blog object they belong to. It implies that a post is not guaranteed to be unique after saving and reloading from the database. Indeed, after loading each blog objects will have their own posts objects even if some posts were shared before saving. Shared objects before saving will be duplicated for each root objects after loading.
4.6 Configure an External Mongo Database [Optional]

We can declare posts as root objects meaning that a post can be saved independently from a blog. It implies that saved blogs have a reference to a TBPost object. This would preserve posts sharing between blog objects.

However, not all objects should be root objects. 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.

**Post as Root = Uniqueness**

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:

```small
TBPost class >> isVoyageRoot
   "Indicates that instances of this class are top level documents 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.

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

```small
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.

4.6 **Configure an External Mongo Database [Optional]**

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. This is not mandatory to do it. Even if you do it, we encourage you to continue to work with a memory database afterwards.
Data Persistence using Voyage and Mongo

Installing Mongo

Regardless of your operating system (Linux, MacOS or Windows), you can install a local Mongo server on your machine (cf. https://www.mongodb.com). This is useful to test your application without requiring an internet connection. Instead directly installing Mongo, we suggest to install Docker (https://www.docker.com) on your machine and execute a Mongo container using the following command line:

```
docker run --name mongo -p 27017:27017 -d mongo
```

**Note** The running Mongo server must not use authentication (it is not the case with the default installation) because the new SCRAM authentication mechanism used by Mongo 3.0 is currently not supported by Voyage.

Some useful Docker commands:

- # to stop your Mongo docker container
  docker stop mongo

- # to re-start your container
  docker start mongo

- # to delete your container (it must be stopped before)
  docker rm mongo

Connecting a Local Mongo Server

Once installed, you can connect to a Mongo server directly from Pharo. We define the method named `initializeLocalhostMongoDB` to establish the connection to the local Mongo server (localhost, default port) and access the database named 'tinyblog'.

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

Reset the class to set a new connection to the database.

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

Now, if you recreate demo posts, they are automatically saved into your local Mongo database:

```
TBBlog reset.
TBBlog createDemoPosts
```
In Case of Trouble

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

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

You can also do it in command line when `mongod` is running with:

```bash
mongo tinyblog --eval "db.dropDatabase()"
```

or by connecting to the docker container it is running in:

```bash
docker exec -it mongo bash -c 'mongo tinyblog --eval "db.dropDatabase()"
```

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:

```bash
VORepository current reset
```

4.7 Conclusion

Voyage proposes a nice API to transparently manage storage of objects either into memory or in a document database. Application data are now saved into a database and we are ready to build the web user interface.
In this chapter, we will setup Seaside and build our first Seaside component. In the next chapters, we will develop the public part of TinyBlog, then the authentication system, followed by the administration part reserved to blog administrators.

All along, we will define Seaside components http://www.seaside.st. A reference book is available online http://book.seaside.st and the firsts chapters may help you and be a great companion of this tutorial book.

All the following work is independent of Voyage and the Mongo database. As usual, you can download the code of previous chapters as explained in the last chapter.

![Seaside Control Panel]

**Figure 5-1** Starting the Seaside server.
5.1 **Starting Seaside**

Seaside should be already loaded in your PharoWeb image. If not, please refer to the loading chapter.

There are two ways to start Seaside. The first one consists in executing the following snippet:

```smalltalk
ZnZincServerAdaptor startOn: 8080.
```

The second one uses the graphical tool named ”Seaside Control Panel” (World Menu>Tools>Seaside Control Panel). In the contextual menu (right clic) of this tool, select ”add adaptor...” and add a server of type `ZnZincServerAdaptor`, then define the port number (e.g. 8080) it should run on (cf. Figure 5-1). By opening a web browser on the URL `http://localhost:8080`, you should see the Seaside home page as displayed on Figure 5-2.

5.2 **Bootstrap for Seaside**

The Bootstrap library is directly accessible from Pharo and Seaside. The repository and the documentation of Bootstrap for Pharo is available there: http://smalltalkhub.com/#!/~TorstenBergmann/Bootstrap. But it is already loaded into the PharoWeb image we are using with this book.

You can also find a public demonstration here: http://pharo.pharocloud.com/bootstrap or directly browse the examples locally in your browser by clicking...
5.3 Define our Application Entry Point

**Figure 5-3** Browsing the Seaside Bootstrap Library.

on the bootstrap link in the list of applications hosted by Seaside or directly enter this URL http://localhost:8080/bootstrap. You should see Bootstrap examples as shown in Figure 5-3.

By clicking on the Examples link at the bottom of the page, you can see both Bootstrap graphical elements and the Seaside code needed to obtain them (cf. Figure 5-4).

### 5.3 Define our Application Entry Point

Create a class named `TBApplicationRootComponent` which will be the entry point of the application.

```small
WAComponent subclass: #TBApplicationRootComponent
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'TinyBlog-Components'
```

We register the TinyBlog application into the Seaside application server by defining the `initialize` class method into the 'initialization' protocol. We also integrate dependencies to the Bootstrap framework (CSS and JS files will be embedded in the application).
First Steps with Seaside

TBApplicationRootComponent class >> initialize
  "self initialize"
  | app |
  app := WAAdmin register: self asApplicationAt: 'TinyBlog'.
  app
    addLibrary: JQDeploymentLibrary;
    addLibrary: JQUiDeploymentLibrary;
    addLibrary: TBSDeploymentLibrary

Once declared, you should execute this method with TBApplicationRootComponent initialize. Indeed, class-side initialize methods are executed at loading-time of a class but since the class already exists, we must execute it by hand.

We also add a method named canBeRoot to specify that TBApplicationRootComponent is not a simple Seaside component but a complete application. This component will be automatically instantiated when a user connects to the application.

TBApplicationRootComponent class >> canBeRoot
  ^ true

You can verify that your application is correctly registered into Seaside by connecting to the Seaside server through your web browser, click on "Browse the applications installed in your image" and then see that TinyBlog appears in the list as illustrated on Figure 5-5. Alternatively, you can visit http://localhost:8080/TinyBlog.

Figure 5-4  A Bootstrap element and its code.
5.4 First Simple Rendering

Let’s add an instance method named `renderContentOn:` in rendering protocol to make our application displaying something.

```small
TBApplicationRootComponent >> renderContentOn: html
   html text: 'TinyBlog'
```

If you open http://localhost:8080/TinyBlog in your web browser, the page should look like the one on Figure 5-6.

You can customize the web page header and declare it as HTML 5 compliant by redefining the `updateRoot:` method.
Figure 5-7  Main components of TinyBlog (public view).

```
TBAplicationRootComponent >> updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    anHtmlRoot beHtml5.
    anHtmlRoot title: 'TinyBlog'
```

The `title:` message is responsible for setting the page title, as can be seen in your web browser's title bar. The `TBAplicationRootComponent` component is the root component of our application. It will not display a lot of things. In the following, it will contain and display other components. For example, a component to display posts to the blog readers, a component to administrate the blog and its posts, ...

5.5 Architecture

We are now ready to define the visual components of our web application.

Overview of TinyBlog

Figure 6-2 shows an overview of them and their responsibilities while Figure 5-8 shows the general architecture of our application and the relations between those components.
Description of the Main Components

To ease your understanding of the incremental development of this application, Figure 5-8 describes the targeted architecture.

- **ApplicationRootComponent** is the entry point registered into Seaside. This component contains components inheriting from the abstract class ScreenComponent.

- **ScreenComponent** is the root of the components used to build the public and administration view of the application. It is composed of a header.

- **PostsListComponent** is the main component that displays the posts. It is composed of instances of PostComponent and manages categories.

- **AdminComponent** is the main component of the administration view. It is composed of a report component (instance of PostsReport) built using Magritte.

5.6 Conclusion

We are now ready to start the development of the described components. In the next chapters, we guide you linearly to develop those components. If you feel lost at some point, we invite you to come back on this architecture overview to better understand what we are developing.
In this chapter, we build the public view of TinyBlog that displays the posts of the blog. Figure 6-1 shows the components we will work on during this chapter. If you feel lost at any moment, please refer to it.

Before starting, you can load the code of previous chapters as described in the last chapter of this book.

6.1 **Visual Components**

Figure 6-2 shows the visual components we will define in this chapter and where they will be displayed.

![Diagram of Component Architecture of the Public View](image)

**Figure 6-1** Component Architecture of the Public View (opposed to the Administration View).
Web Components for TinyBlog

Figure 6-2  Visual Components of TinyBlog.

The TBScreenComponent component

All components contained in TBApplicationRootComponent will be subclasses of the abstract class TBScreenComponent. This class allows us to factorize shared behavior between all our components.

```plaintext
WAClassComponent subclass: #TBScreenComponent
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'TinyBlog-Components'
```

All components need to access the model of our application. Therefore, in the 'accessing' protocol, we add a blog method that returns the current instance of TBBlog (the singleton). In the future, if you want to manage multiple blogs, you will modify this method and return the blog object it has been configured with.

```plaintext
TBScreenComponent >> blog
    "Return the current blog. In the future we will ask the session to return the blog of the currently logged in user."
    ^ TBBlog current
```

Let’s define a method renderContentOn: on this new component that temporarily displays a message. If you refresh your browser, nothing appears because this new component is not displayed at all yet.

```plaintext
TBScreenComponent >> renderContentOn: html
    html text: 'Hello from TBScreenComponent'
```
6.2 Using the TBScreenComponent component

In the final architecture, TBScreenComponent is an abstract component and should not be used directly. Nevertheless, we will use it temporarily while developing other components.

Let's add an instance variable main in TBApplicationRootComponent class. We obtain the situation described in Figure 6-3.

We initialize this instance variable in the initialize method with a new instance of TBScreenComponent.

We make the TBApplicationRootComponent to render this sub-component.

We do not forget to declare that the object contained in main instance variable is a sub-component of TBApplicationRootComponent by redefining the children method.

Figure 6-4 shows the result that you should obtain in your browser. Currently, there is only the text: Hello from TBScreenComponent displayed by the TBScreenComponent sub-component. (voir figure 6-4).

6.3 Pattern of Component Definition

We will often use the same following steps:

- first, we define a class and the behavior of a new component;
Web Components for TinyBlog

Figure 6-4 First visual rendering of TBScreenComponent.

- then, we reference it from an existing component that uses it;
- and we express the composite/sub-component relationship by redefining the children method.

6.4 Populating the Blog

You can inspect the blog object returned by TBBlog current and verify that it contains some posts. You can also do it simply as:

```
TBBlog current allBlogPosts size
```

If it does not, execute:

```
TBBlog createDemoPosts
```

6.5 Definition of TBHeaderComponent

Let’s define a component named TBHeaderComponent that renders the common header of all pages of TinyBlog. This component will be inserted on the top of all components such as TBPostsListComponent. We use the pattern described above: define the class of the component, reference it from its enclosing component and redefine the children method.

Here the class definition:

```
WAComponent subclass: #TBHeaderComponent
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'TinyBlog-Components'
```

6.6 Usage of TBHeaderComponent

Remember that TBScreenComponent is the (abstract) root of all components in our final architecture. Therefore, we will introduce our header into TBScreenComponent so that all its subclasses will inherit it. Since, it is not desirable to instantiate the TBHeaderComponent each time a component is called, we store the header in an instance variable named header.
6.7 Composite-Component Relationship

We initialize it in the initialize method categorized in the 'initialization' protocol:

```smalltalk
TBScreenComponent >> initialize
    super initialize.
    header := self createHeaderComponent
```

Note that we use a specific method named createHeaderComponent to create the instantiate the header component. Redefining this method makes it possible to completely change the header component that is used. We will use that to display a different header component for the administration view.

6.7 Composite-Component Relationship

In Seaside, sub-components of a component must be returned by the composite when sending it the children message. So, we must define that the TBHeaderComponent instance is a children of the TBScreenComponent component in the Seaside component hierarchy (and not in the Pharo classes hierarchy). We do so by specializing the method children. In this example, it returns a collection of one element which is the instance of TBHeaderComponent referenced by the header instance variable.

```smalltalk
TBScreenComponent >> children
    ^ { header }
```

6.8 Render an header

In the renderContentOn: method ('rendering' protocol), we can now display the sub-component (the header):

```smalltalk
TBScreenComponent >> renderContentOn: html
    html render: header
```

If you refresh your browser, nothing appears because the TBHeaderComponent has no rendering. Let’s add a renderContentOn: method on it that displays a Bootstrap navigation header:
Web Components for TinyBlog

**Figure 6-5** TinyBlog with a Bootstrap header.

```
TBHeaderComponent >> renderContentOn: html
  html tbsNavbar beDefault; with: [
    html tbsContainer: [
      self renderBrandOn: html
    ]
  ]

TBHeaderComponent >> renderBrandOn: html
  html tbsNavbarHeader: [
    html tbsNavbarBrand
    url: self application url;
    with: 'TinyBlog'
  ]
```

Your browser should now display what is shown on Figure 6-5. As usual in Bootstrap navigation bar, the link on the title of the application (tbsNavbarBrand) enable users to go back to home page of the application.

**Possible Enhancements**

The blog name should be customizable using an instance variable in the TB-Blog class and the application header component should display this title.
6.9 List of Posts

Let’s create a TBPostsListComponent inheriting from TBScreenComponent to display the list of all posts. Remember that we speak about the public access to the blog here and not the administration interface that will be developed later.

```smalltalk
TBScreenComponent subclass: #TBPostsListComponent
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'TinyBlog-Components'
```

We can now modify TBApplicationRootComponent, the main component of the application, so that it displays this new component as shown in figure 6-6. To achieve this, we modify its initialize method:

```smalltalk
TBApplicationRootComponent >> initialize
    super initialize.
    main := TBPostsListComponent new

TBApplicationRootComponent >> main: aComponent
    main := aComponent
```

We add a setter method named main: to dynamically change the sub-component to display but by default it is an instance of TBPostsListComponent.

```smalltalk
TBApplicationRootComponent >> main: aComponent
    main := aComponent
```

We now add a temporary renderContentOn: method (in the 'rendering' protocol) on TBPostsListComponent to test during development (cf. Figure 6-7). In this method, we call the renderContentOn: of the super-class which renders the header component.

```smalltalk
TBPostsListComponent >> renderContentOn: html
    super renderContentOn: html.
    html text: 'Blog Posts here !!!'
```

If you refresh TinyBlog in your browser, you should now see what is shown in figure 6-7.

Figure 6-6  The ApplicationRootComponent uses PostsListComponent.
Figure 6-7  TinyBlog displaying a basic posts list.

Figure 6-8  Using PostComponents to display each post.

6.10  The PostComponent

Now we will define TBPostComponent to display the details of a post. Each post will be graphically displayed by an instance of TBPostComponent which will show the post title, its date and its content as shown in figure 6-8.

```smalltalk
WACComponent subclass: #TBPostComponent
    instanceVariableNames: 'post'
    classVariableNames: ''
    package: 'TinyBlog-Components'

TBPostComponent >> initialize
    super initialize.
    post := TBPost new
```
6.11 Display Posts

The `renderContentOn:` method defines the HTML rendering of a post.

``` Smalltalk
TBPostComponent >> renderContentOn: html
    html heading level: 2; with: self title.
    html heading level: 6; with: self date.
    html text: self text
```

About HTML Forms

In a future chapter on the administration view, we will show how to use Magritte to add descriptions to model objects and then use them to automatically generate Seaside components. This is powerful and free developers to manually describe forms in Seaside.

To give you a taste of that, here the equivalent code as above using Magritte:

``` Smalltalk
TBPostComponent >> renderContentOn: html
    "DON'T WRITE THIS YET"
    html render: post asComponent
```

6.11 Display Posts

Before displaying available posts in the database, you should check that your blog contains some posts:

``` Smalltalk
TBBlog current allBlogPosts size
```

If it contains no posts, you can recreate some:

``` Smalltalk
TBBlog createDemoPosts
```

Now, we just need to modify the `TBPostsListComponent >> renderContentOn:` method to display all visible posts in the database:

``` Smalltalk
TBPostsListComponent >> renderContentOn: html
    super renderContentOn: html.
    self blog allVisibleBlogPosts do: [ :p |
        html render: (TBPostComponent new post: p) ]
```

Refresh you web browser and you should get an error.
6.12 Debugging Errors

By default, when an error occurs in a web application, Seaside returns an HTML page with the error message. You can change this message or during development, you can configure Seaside to open a debugger directly in Pharo IDE. To configure Seaside, just execute the following snippet:

```small
(WAAdmin defaultDispatcher handlerAt: 'TinyBlog')
    exceptionHandler: WADebugErrorHandler
```

Now, if you refresh the web page in your browser, a debugger should open on Pharo side. If you analyze the stack, you should see that we forgot to define the following method:

```small
TBPostComponent >> post: aPost
    post := aPost
```

You can define this method in the debugger using the Create button. After that, press the Proceed button. The web application should now correctly renders what is shown in Figure 6-9.

6.13 Displaying the List of Posts with Bootstrap

Let’s use Bootstrap to make the list of posts more beautiful using a Bootstrap container thanks to the message tbsContainer::
6.14  **Instantiating Components in renderContentOn:**

We explained that the children method of a component should return its sub-components. Indeed, before executing the renderContentOn: method of a composite, Seaside needs to retrieve all its sub-components and their state. However, if sub-components are instantiated in the renderContentOn: method of the composite (such as in TBPostsListComponent>>renderContentOn:), it is not needed that children returns those sub-components.

Note that, instantiating sub-components in the rendering method is not a good practice since it increases the loading time of the web page.

If we would store all sub-components that display posts, we should add an instance variable postComponents.

``` Smalltalk
TBPostsListComponent >> initialize
super initialize.

postComponents := OrderedCollection new
```

Initialize it with posts.

``` Smalltalk
TBPostsListComponent >> postComponents
postComponents := self readSelectedPosts collect:
  [ :each | TBPostComponent new post: each ].

^ postComponents
```

Redefine the children method and of course render these sub-components in renderContentOn::

``` Smalltalk
TBPostsListComponent >> children
^ self postComponents, super children
```

We do not do this in TinyBlog because it makes the code more complex.
6.15 Conclusion

In this chapter, we developed a Seaside component that renders a list of posts. In the next chapter, we will improve this by displaying posts’ categories.

Notice that we did not care about web requests or the application state. A Seaside programmer only define components and compose them as we would do in desktop applications.

A Seaside component is responsible of rendering itself by redefining its renderContentOn: method. It should also returns its sub-components (if no instantiated during each rendering) by redefining the children method.
In this chapter, we add the possibility to sort posts in a category. Figure 7-1 shows you on which components we will work in this chapter.

You can find instructions to load the code of previous chapter in Chapter 11.

### 7.1 Displaying Posts by Category

Posts are sorted by a category. If no category is specified, posts are sorted in a special category called "Unclassified". To manage a list of categories, we will define a component named `TBCategoriesComponent`.

![Figure 7-1](image)

L’architecture des composants de la partie publique with categories.
Displaying Categories

We need a component to display a list of categories defined in the blog. This component should support the selection of one category. This component should be able to communicate with the component TBPostsListComponent to give it the currently selected category. Figure 7-1 described the situation.

Remember that a category is simply expressed as a string in the model we defined in Chapter 2 and how the following test illustrates it:

```smalltalk
testAllBlogPostsFromCategory
  self assert: (blog allBlogPostsFromCategory: 'First Category')
  size equals: 1
```

Component Definition

Let us define a new component named TBCategoriesComponent. It keeps a sorted collection of string representing each category as well as a reference to the component managing the post list.

```smalltalk
WAClassDef subclass: #TBCategoriesComponent
  instanceVariableNames: 'categories postsList'
  classVariableNames: '
  package: 'TinyBlog-Components'

TBCategoriesComponent >> categories
  ^ categories

TBCategoriesComponent >> categories: aCollection
  categories := aCollection asSortedCollection

TBCategoriesComponent >> postsList: aComponent
  postsList := aComponent

TBCategoriesComponent >> postsList
  ^ postsList

We define a creation method as a class method.
TBCategoriesComponent class >> categories: categories postsList: aTBScreen
  ^ self new categories: categories; postsList: aTBScreen
```

From the Post List

In the class TBPostsListComponent, we need to add an instance variable to store the current category.
7.2 Category Rendering

[TBScreenComponent subclass: #TBPostsListComponent]
    instanceVariableNames: 'currentCategory'
    classVariableNames: '
    package: 'TinyBlog-Components'

We define its associated accessors.

[TBPostsListComponent >> currentCategory]
    ^ currentCategory

[TBPostsListComponent >> currentCategory: anObject]
    currentCategory := anObject

The method selectCategory:

We define the method selectCategory: (protocol 'actions') to communicate the current category to the TBPostsListComponent component.

[TBCategoriesComponent >> selectCategory: aCategory]
    postsList currentCategory: aCategory

7.2 Category Rendering

We can now define method for the rendering of the category component on the page. Let us call it renderCategoryLinkOn:with:, we define in particular that clicking on a category will select it as the current one. We use a callback (message callback:). The argument of this message is a block that can contains any Pharo expression. This illustrates how simple is to call function to react to event.

[TBCategoriesComponent >> renderCategoryLinkOn: html with: aCategory]
    html tbsLinkifyListGroupItem
        callback: [ self selectCategory: aCategory ];
        with: aCategory

The method renderContentOn: of TBCategoriesComponent is simple: we iterate on all categories and we display them using Bootstrap.

[TBCategoriesComponent >> renderContentOn: html]
    html tbsListGroup: [
        html tbsListGroupItem
            with: [ html strong: 'Categories' ].
        categories do: [ :cat |
            self renderCategoryLinkOn: html with: cat ]
    ]

We are nearly there. We need to display the list of categories and update the posts based on the current category.
7.3 **Updating Post List**

Now we should update the list of posts. We modify the rendering method of the component TBPostsListComponent.

The method readSelectedPosts collects the posts to be displayed. It filters them based on the current category. When the current category is nil, it means that the user did not select yet a category. Therefore we display all the posts. When the current category is something else than nil, the user selected a category and the application display the corresponding posts.

```plaintext
TBPostsListComponent >> readSelectedPosts
^ self currentCategory
  ifNil: [ self blog allVisibleBlogPosts ]
  ifNotNil: [ self blog allVisibleBlogPostsFromCategory: self currentCategory ]
```

We modify now the method responsible of the post list rendering:

```plaintext
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html render: (TBCategoriesComponent
categories: (self blog allCategories)
postsList: self).
  html tbsContainer: [
    self readSelectedPosts do: [ :p |
      html render: (TBPostComponent new post: p) ] ]
```

An instance of the component TBCategoriesComponent is added to the page and allows one to select the current category (see Figure 7-2).

As previously explained, a new instance of TBCategoriesComponent is created each time the component TBPostsListComponent is rendered, therefore it is not mandatory to add it to the children sublist of the component.

**Possible Enhancements**

Hardcoding class name and the creation logic of categories and posts is not really optimal. Propose some solution.

7.4 **Look and Layout**

We will not place better the component TBPostsListComponent using a more ‘responsive’ design (as shown in Figure 7-3). It means that the CSS style should adapt the component to the available space.

Components are placed in a Bootstrap container then positioned on a line with two columns. Column dimension is determined based on the viewport and resolution of the device used. The 12 columns of Bootstrap are distributed over the category and post lists.
7.4 Look and Layout

In the case of a low resolution, the list of categories is placed above the post list (each element using 100% of the container width).

```small
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html tbsContainer: [html tbsRow showGrid; with: [html tbsColumn extraSmallSize: 12; smallSize: 2; mediumSize: 4; with: [html render: (TBCategoriesComponent categories: (self blog allCategories) postsList: self)].html tbsColumn extraSmallSize: 12; smallSize: 10; mediumSize: 8; with: [self readSelectedPosts do: [:p | html render: (TBPostComponent new post: p)]]]]]
```

You should obtain a situation close to the one presented in Figure 7-3.
When one selects a category, the post list is updated. However, the selected category is not selected. We modify the following method to address this point.

```plaintext
TBCategoriesComponent >> renderCategoryLinkOn: html with: aCategory
  html tbsLinkifyListGroupItem
  class: 'active' if: aCategory = self postsList currentCategory;
  callback: [ self selectCategory: aCategory ];
  with: aCategory
```

Even if the code works, we cannot keep the method `renderContentOn:` in such state. It is far too long and not reusable. Propose a solution.

### 7.5 Modular Code with Small Methods

Here is our solution to the previous problem. To ease reading and future reuse, we start to define component creation methods.

```plaintext
TBPostsListComponent >> categoriesComponent
  ^ TBCategoriesComponent
    categories: self blog allCategories
    postsList: self

TBPostsListComponent >> postComponentFor: aPost
  ^ TBPostComponent new post: aPost
```
Another Pass

We continue to cut this method in other smaller methods. We create one method for each of the elementary tasks.

```smalltalk
TBPostsListComponent >> basicRenderCategoriesOn: html
    html render: self categoriesComponent

TBPostsListComponent >> basicRenderPostsOn: html
    self readSelectedPosts do: [:p | html render: (self postComponentFor: p)]
```

Then we use such tasks to simplify the method `renderContentOn:`.

```smalltalk
TBPostsListComponent >> renderContentOn: html
    super renderContentOn: html.
    html
        tbsContainer: [: html tbsRow
            showGrid;
            with: [: html tbsColumn
                extraSmallSize: 12;
                smallSize: 2;
                mediumSize: 4;
                with: [: html render: self categoriesComponent ].
            html tbsColumn
                extraSmallSize: 12;
                smallSize: 10;
                mediumSize: 8;
                with: [: self readSelectedPosts
do: [: :p | html render: (self postComponentFor: p) ] ]
        ] ] ]
```
Managing Categories

Figure 7-4  Final TinyBlog Public UI.

```javascript
TBPostsListComponent >> renderPostColumnOn: html
   html tbsColumn
      extraSmallSize: 12;
      smallSize: 10;
      mediumSize: 8;
      with: [ self basicRenderPostsOn: html ]
```

The final application is showing in Figure 7-4.

7.6 Conclusion

We defined an interface for our blog using a set of components each specifying its own state and responsibility. Many web applications are built the same way reusing components. You have the foundation to build more advanced web application.

In the next chapter, we show how to manage authentication to access the post admin part of our application.

Possible Enhancements

As exercise you can:

- sort category alphabetically, or
- add a link named 'All' in the category list to display all the posts.
In this chapter we will develop a traditional scenario: the user should login to access to the administration part of the application. He does it using a login and password.

Figure 8-1 shows the architecture that we will reach in this chapter.

Let us start to put in place a first version that allows one to navigate between the part of TinyBlog rendered by the component TBPostsListComponent and a first draft of the administration component as shown in Figure 8-2. This illustrates how to invoke a component.

**Figure 8-1** Authentication flow.
In the following we will build and integrate a component managing the login based on modal interaction. This will illustrate how we can elegantly map filed inputs to instance variables of a component.

Finally we will show how the user information is stored into the current session.

### 8.1 A Simple Admin Component (v1)

Let us define a really super simple administration component. This component inherits from the class TBScreenComponent as mentioned in previous chapters and illustrated in Figure 8-1.

```
TBScreenComponent subclass: #TBAdminComponent
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'TinyBlog-Components'
```

We define a first version of the rendering method to be able to test our approach.

```
TBAdminComponent >> renderContentOn: html
    super renderContentOn: html.
    html tbsContainer: [
        html heading: 'Blog Admin'.
        html horizontalRule ]
```

### 8.2 Adding ‘admin’ Button

We add now a button in the header of the site (component TBHeaderComponent) so that the user can access to the admin as shown in Figure 8-2. To do so, we modify the existing components: TBHeaderComponent (header) et TBPostsListComponent (public part).

Let us add a button in the header:

```
TBHeaderComponent >> renderContentOn: html
    html tbsNavbar beDefault; with: [
        html tbsContainer: [
            self renderBrandOn: html.
            self renderButtonsOn: html
        ]
    ]
```

```
TBHeaderComponent >> renderButtonsOn: html
    self renderSimpleAdminButtonOn: html
```

```
TBHeaderComponent >> renderSimpleAdminButtonOn: html
    html form: [
        html tbsNavBarButton
        tbsPullRight;
    ]
```
8.2 Adding 'admin' Button

Figures 8-2 and 8-3 illustrate the admin part of the TinyBlog application. Figure 8-2 shows a simple link to the admin part, while Figure 8-3 displays the header with an admin button.

```html
with: [
  html tbsGlyphIcon iconListAlt.
  html text: ' Admin View'
]
```

When you refresh the web browser, the admin button is present but it does not have any effect (See Figure 8-3).

We should define a callback on this button (message callback:) to replace the current component (TBPostsListComponent) by the administration component (TBAdminComponent).
8.3 **Header Revision**

Let us revise the definition of TBHeaderComponent by adding a new instance variable named `component` to store and access to the current component (either post list or admin component). This will allow us to access to the component from the header.

```vbnet
WAComponent subclass: #TBHeaderComponent
    instanceVariableNames: 'component'
    classVariableNames: ''
    package: 'TinyBlog-Components'

TBHeaderComponent >> component: anObject
    component := anObject

TBHeaderComponent >> component
    ^ component

We add a new class method.

```vbnet
TBHeaderComponent class >> from: aComponent
    ^ self new
    component: aComponent;
    yourself
```

8.4 **Admin Button Activation**

We modify the component instantiation in TBScreenComponent method to pass the component which will be under the header.

```vbnet
TBScreenComponent >> createHeaderComponent
    ^ TBHeaderComponent from: self

Note that the method `createHeaderComponent` is defined in the superclass TBScreenComponent and it is applicable to all the subclasses.

We can add now the callback on the button:

```vbnet
TBHeaderComponent >> renderSimpleAdminButtonOn: html
    html form: [
        html tbsNavbarButton
            tbsPullRight;
            callback: [ component goToAdministrationView ];
            with: [
                html tbsGlyphIcon iconListAlt.
                html text: ' Admin View' ]
    ]

We just need to define the method `goToAdministrationView` on the component TBPostsListComponent:

```vbnet
TBPostsListComponent >> goToAdministrationView
    self call: TBAdminComponent new
Before clicking on the admin button, you should renew the current session by clicking on 'New Session': it will recreate the component TBHeaderComponent.

You should get the situation presented in Figure 8-4. The 'Admin' button allows one to access the admin part v1.

Pay attention not to click twice on the admin button because we do not manage it yet for the admin part. We will replace it by a Disconnect button.

8.5 'disconnect' Button Addition

When we display the admin part, we will replace the header component by a new one. This new header will display a disconnect button.

Let us define this new header component:

```smalltalk
TBHeaderComponent subclass: #TBAadminHeaderComponent
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'TinyBlog-Components'

TBAadminHeaderComponent >> renderButtonsOn: html
  html form: [ self renderDisconnectButtonOn: html ]
```

The TBAadminComponent component must use this header:

```smalltalk
TBAadminComponent >> createHeaderComponent
  ^ TBAadminHeaderComponent from: self
```
Now we can specialize our new admin header to display a disconnect button.

```plaintext
TBAadminHeaderComponent >> renderDisconnectButtonOn: html
  html tbsNavBarButton
  tbsPullRight;
  callback: [ component goToListView ];
  with: [
    html text: 'Disconnect '.
    html tbsGlyphIcon iconLogout ]
```

What is see is that the message answer gives back the control to the component that calls it. So we go back the post list.

Reset the current session by clicking on 'New Session'. Then you can click on the 'Admin' button, you should see now the admin v1 display itself with a 'Disconnect' button. This button allows on to go back the public part as shown in Figure 8-2.

**call:/answer: Notion**

When you study the previous code, you see that we use the call:/answer: mechanism of Seaside to navigate between the components TBPostsListComponent and TBAadminComponent.

The message call: replaces the current component with the one passed in argument and gives it the flow of control. The message answer: returns a value to this call and gives back the flow of control to the calling argument. This mechanism is really powerful and elegant. It is explained in the vidéo 1 of week 5 of the Pharo Mooc (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_5.html).

### 8.6 Modal Window for Authentication

Let us develop now a authentication component that when invoked will open a dialog box to request the login and password. The result we want to obtain is shown in Figure 8-5.

There are are some libraries of components ready to be used. For example, the Heimdal project available at http://www.github.com/DuneSt/ offers an authentication component or the Steam project https://github.com/guillep/steam offers ways to interrogate google ou twitter accounts.

**Authentication Component Definition**

We define a new subclass of WAComponent and its accessors. This component contains a login, a password and a component which invoked it to access to
8.6 Modal Window for Authentication

Figure 8-5  Authentication component.

the admin part.

WAComponent subclass: #TBAuthenticationComponent
  instanceVariableNames: 'password account component'
  classVariableNames: ''
  package: 'TinyBlog-Components'

TBAuthenticationComponent >> account
  ^ account

TBAuthenticationComponent >> account: anObject
  account := anObject

TBAuthenticationComponent >> password
  ^ password

TBAuthenticationComponent >> password: anObject
  password := anObject

TBAuthenticationComponent >> component
  ^ component

TBAuthenticationComponent >> component: anObject
  component := anObject

The instance variable component will be initialized by the following class method: classe suivante:

TBAuthenticationComponent class >> from: aComponent
  ^ self new
    component: aComponent;
    yourself
8.7 **Authentication Component Rendering**

The following method `renderContentOn:` defines the contents of a dialog box with the ID `myAuthDialog`. This ID will be used to select the component that should be made visible when in modal mode.

This dialog box has a header and a body. Note the use of the messages `tbsModal`, `tbsModalBody:`, and `tbsModalContent:` which supports a modal interaction with the component.

```plaintext
TBAuthentificationComponent >> renderContentOn: html
    html tbsModal
        id: 'myAuthDialog';
        with: [
            html tbsModalDialog: [
                html tbsModalContent: [
                    self renderHeaderOn: html.
```

The header displays a button to close the dialog box and a title with large fonts. Note that you can also use the ESC key to close the modal window box.

```plaintext
TBAuthentificationComponent >> renderHeaderOn: html
    html
        tbsModalHeader: [
            html tbsModalCloseIcon.
            html tbsModalTitle
                level: 4;
                with: 'Authentication'
        ]
```

The body of the component displays the input field for the login identifier, password and some buttons.

```plaintext
TBAuthentificationComponent >> renderBodyOn: html
    html
        tbsModalBody: [
            html tbsForm: [
                self renderAccountFieldOn: html.
                self renderPasswordFieldOn: html.
                html tbsModalFooter: [ self renderButtonsOn: html ]
            ]
        ]
```

The method `renderAccountFieldOn:` shows how the value of an input field is passed and stored in an instance variable of a component when the user finishes its input.

The parameter of the `callback` message is a bloc which takes as argument the value of the input field.
8.8 Authentication Component Integration

```tcl
TBAuthentificationComponent >> renderAccountFieldOn: html
   html
      tbsFormGroup: [ html label with: 'Account'.
         html textInput
            tbsFormControl;
            attributeAt: 'autofocus' put: 'true';
            callback: [ :value | account := value ];
            value: account ]

The same process is used for the password.

TBAuthentificationComponent >> renderPasswordFieldOn: html
   html tbsFormGroup: [ html label with: 'Password'.
         html passwordInput
            tbsFormControl;
            callback: [ :value | password := value ];
            value: password ]

Finally in the following renderContentOn: method, two buttons are added
at the bottom of the modal window. The 'Cancel' button which allows one
to close the window using the attribute 'data-dismiss' and the 'SignIn' but-
tton which sends the validate using a callback.

The enter key is bound to the 'SignIn' button activation when using the
method tbsSubmitButton. This method sets the 'type' attribute to 'submit'.

TBAuthentificationComponent >> renderButtonsOn: html
   html tbsButton
      attributeAt: 'type' put: 'button';
      beDefault;
      value: 'Cancel'.
   html tbsSubmitButton
      bePrimary;
      callback: [ self validate ];
      value: 'SignIn'

In the validate method, we simply send a message to the main component
giving it the information entered by the user.

TBAuthentificationComponent >> validate
   ^ component tryConnectionWithLogin: self account andPassword: self password
```

8.8 Authentication Component Integration

To integrate our authentication component, we modify the Admin button of
the header component (TBHeaderComponent) as follows:
The method `renderModalLoginButtonOn:` starts by rendering the component `TBAuthentificationComponent` within this web page. This component is created during each display and it does not have to be returned by the `children` method. In addition, we add a 'Login' button with a icon lock. When the user clicks on this button, the modal dialog identified with the ID `myAuthDialog` will be displayed.

Reloading the TinyBlog page, you should see now a 'Login' button in the header (button that will pop up the authentication we just developed) as illustrated by Figure 8-5.

### 8.9 Naively Managing Logins

When you click on the 'SignIn' button you get an error. Using the Pharo debugger, you can see that we should define the method `tryConnectionWithLogin:andPassword:` on the component `TBPostsListComponent` since it is the one sent by the callback of the button.

```plaintext
TBPostsListComponent >> tryConnectionWithLogin: login andPassword: password

(login = 'admin' and: [ password = 'topsecret' ]) ifTrue: [ self goToAdministrationView ]
ifFalse: [ self loginErrorOccurred ]
```

For the moment we store directly the login and password in the method and this is not really a good practice.

### 8.10 Managing Errors

We defined the method `goToAdministrationView`. Let us add the method `loginErrorOccured` and a mechanism to display an error message when the user does not use the correct identifiers as shown in Figure 8-6.

For this we will add a new instance variable `showLoginError` that represents the fact that we should display an error.
8.11 Modeling the Admin

TBScreenComponent subclass: #TBPostsListComponent
  instanceVariableNames: 'currentCategory showLoginError'
  classVariableNames: '
  package: 'TinyBlog-Components'

The method loginErrorOccurred specifies that an error should be displayed.

TBPostsListComponent >> loginErrorOccurred
  showLoginError := true

We add a method to test this state.

TBPostsListComponent >> hasLoginError
  ^ showLoginError ifNil: [ false ]

We define also an error message.

TBPostsListComponent >> loginErrorMessage
  ^ 'Incorrect login and/or password'

We modify the method renderPostColumnOn: to perform a specific task to handle the errors.

TBPostsListComponent >> renderPostColumnOn: html
  html tbsColumn
    extraSmallSize: 12;
    smallSize: 10;
    mediumSize: 8;
    with: [
      self renderLoginErrorMessageIfAnyOn: html.
      self basicRenderPostsOn: html ]

The method renderLoginErrorMessageIfAnyOn: displays if necessary an error message. It sets the instance variable showLoginError so that we do not display the error indefinitely.

TBPostsListComponent >> renderLoginErrorMessageIfAnyOn: html
  self hasLoginError ifTrue: [
    showLoginError := false.
    html tbsAlert
      beDanger ;
      with: self loginErrorMessage
  ]

8.11 Modeling the Admin

We do not want to store the administrator identifiers in the code as we did previously. We revise this now and will store the identifiers in a model: a class Admin.
Let us start to enrich our TinyBlog model with the notion of administrator. We define a class named `TBAdministrator` characterized by its pseudo, login and password.

```smalltalk
Object subclass: #TBAdministrator
  instanceVariableNames: 'login password'
  classVariableNames: ''
  package: 'TinyBlog'

TBAdministrator >> login
  ^ login

TBAdministrator >> login: anObject
  login := anObject

TBAdministrator >> password
  ^ password

TBAdministrator >> password: anObject
  password := SHA256 hashMessage: anObject

We define also a new instance creation method.

```smalltalk
TBAdministrator class >> login: login password: password
  ^ self new
    login: login;
    password: password;
    yourself
```

You can verify that the model works by executing the following expression:

```smalltalk
luc := TBAdministrator login: 'luc' password: 'topsecret'.
```
8.12 Blog admin

We decide for simplicity that a blog has one admin. We add the instance variable adminUser and an accessor in the classe TBBlog to store the blog admin.

```smalltalk
Object subclass: #TBBlog
    instanceVariableNames: 'adminUser posts'
    classVariableNames: '
    package: 'TinyBlog'

TBBlog >> administrator
    ^ adminUser
```

We define a default login and password that we use as default. As we will see later, we will modify such attributes and these modified attributes will be saved at the same time that the blog in a database.

```smalltalk
TBBlog class >> defaultAdminPassword
    ^ 'topsecret'

TBBlog class >> defaultAdminLogin
    ^ 'admin'
```

Now we create a default admin.

```smalltalk
TBBlog >> createAdministrator
    ^ TBAdministrator
        login: self class defaultAdminLogin
        password: self class defaultAdminPassword
```

And we initialize the blog to set a default administrateur.

```smalltalk
TBBlog >> initialize
    super initialize.
    posts := OrderedCollection new.
    adminUser := self createAdministrator
```

8.13 Setting a New Admin

We should not recreate the blog:

```smalltalk
TBBlog reset; createDemoPosts
```

We can now modify the admin information as follows:

```smalltalk
|admin|
    admin := TBBlog current administrator.
    admin login: 'luke'.
    admin password: 'thebrightside'.
    TBBlog current save
```

Note that without doing anything, the blog admin information has been saved by Voyage in the database. Indeed the class TBBlog is a Voyage root,
all its attributes are automatically stored in the database when it received the message save.

Possible Enhancements
Define some tests for the extensions by writing new unit tests.

8.14 Integrating the Admin Information

Let us modify the method tryConnectionWithLogin:andPassword: so that it uses the current blog admin identifiers. Note that we are comparing the hash SHA256 of the password since we do not store the password.

```small
TBPostsListComponent >> tryConnectionWithLogin: login andPassword: password
  (login = self blog administrator login and: [
    (SHA256 hashMessage: password) = self blog administrator password ])
  ifTrue: [ self goToAdministrationView ]
  ifFalse: [ self loginErrorOccurred ]
```

8.15 Storing the Admin in the Current Session

With the current setup, when the blog admin wants to navigate between the private and public part, he must reconnects each time. We will simplify this situation but storing the current admin information in the session when the connection is successful.

A session object is given to the each instance of the application. Such session allows on to keep information which are shared and accessible between components.

We will then store the current admin in a session and modify the components to display buttons that support a simplified navigation when the admin is logged.

When he explicitely disconnect or when the session expires, we delete the current session.

Figure 8-7 shows the navigation between the pages of TinyBlog.

8.16 Definition and use of specific session

Let us start to define a subclass of WASession and name it TBSession. We add in this new class an instance variable that stores the current admin.
8.16 Definition and use of specific session

```
WASession subclass: #TBSession
  instanceVariableNames: 'currentAdmin'
  classVariableNames: '
  package: 'TinyBlog-Components'

TBSession >> currentAdmin
  ^ currentAdmin

TBSession >> currentAdmin: anObject
  currentAdmin := anObject

We define a method isLogged allows one to know if the administration is
logged.

TBSession >> isLogged
  ^ self currentAdmin notNil

Now we should indicate to Seaside to use TBSession as the class of the cur-
rent session for our application. This initialization is done in the class method
initialize in the class TBAplicationRootComponent as follows:

TBAplicationRootComponent class >> initialize
  "self initialize"
  | app |
  app := WAAdmin register: self asApplicationAt: 'TinyBlog'.
  app
    preferenceAt: #sessionClass put: TBSession.
    app
      addLibrary: JQDeploymentLibrary;
```

**Figure 8-7** Navigation and identification in TinyBlog.
addLibrary: JQuiDeploymentLibrary;
addLibrary: TBSDeploymentLibrary

Do not forget to execute this expression TBAplicationRootComponent initialize before testing the application.

8.17 Storing the Current Admin

When a connection is successful, we add the admin object to the current session using the message currentAdmin:. Note that the current session is available to every Seaside component via self session.

```smalltalk
TBPotsListComponent >> tryConnectionWithLogin: login andPassword: password
    (login = self blog administrator login and: [SHA256 hashMessage: password] = self blog administrator password)
    ifTrue: [self session currentAdmin: self blog administrator.
               self goToAdministrationView]
    ifFalse: [self loginErrorOccurred]
```

8.18 Simplified navigation

To put in place the simplified navigation we discussed above, we modify the header to display either a login button or a simple navigation button to the admin part without forcing any reconnection. For this we use the session and the fact that we can know if a user is logged.

```smalltalk
TBHeaderComponent >> renderButtonsOn: html
    self session isLogged
    ifTrue: [self renderSimpleAdminButtonOn: html]
    ifFalse: [self renderModalLoginButtonOn: html]
```

You can test this new navigation but first create a new session (‘New Session’ button). One reconnected the admin is added in session. Note that the deconnection button does not work correctly since it does invalidate the session.

8.19 Managing Deconnection

We add a method reset on our session object to delete the current admin, invalidate the current session and redirect to the application entry point.
TBSession >> reset
    currentAdmin := nil.
    self requestContext redirectTo: self application url.
    self unregister.

Now we modify the header deconnection button to send the message reset
to the correct session.

TBAdminHeaderComponent >> renderDisconnectButtonOn: html
    html tbsNavbarButton
        tbsPullRight;
        callback: [ self session reset ];
        with: [
            html text: 'Disconnect '.
            html tbsGlyphIcon iconLogout ]

Now we 'Disconnect' button works the way it should.

8.20 Simplified Navigation to the Public Part

We can add now a button in the header of the admin part to go back to the
public part without being forced to get disconnected.

TBAdminHeaderComponent >> renderButtonsOn: html
    html form: [
        self renderDisconnectButtonOn: html.
        self renderPublicViewButtonOn: html ]

TBAdminHeaderComponent >> renderPublicViewButtonOn: html
    self session isLogged ifTrue: [
        html tbsNavbarButton
            tbsPullRight;
            callback: [ component goToPostListView ];
            with: [
                html tbsGlyphIcon iconEyeOpen.
                html text: ' Public View' ]

Now you can test the navigation. It should correspond to the situation de-
picted by Figure 8-7.

8.21 Conclusion

We put in place an authentication for TinyBlog. We create a reusable modal
component. We made the distinction between component displayed when a
user is connected or not and ease the navigation of a connected user using
session.

We are now ready for the administration part of the application and we will
work on this in the next chapter. We will take advantage of it to show and
advanced aspect: the automatic form generation.
Possible Enhancements

You can:

• Add the admin logging in the header
• Manage multiple admin accounts.
We will now develop the administration part of TinyBlog. In previous chapter, we define Seaside components that interact together and where each component is responsible for its internal state, behavior and its graphical rendering.

In this chapter, we want to show that we can go a step further and generate Seaside components from object descriptions using the Magritte framework. Figure 9-1 shows a part of the result we will obtain, the other part being post edition.

Figure 9-2 shows a survey of the architecture that we will develop in this chapter.

9.1 **Describing Domain Data**

Magritte is a library that allows one to generate various representations once the objects are described. Coupled with Seaside, Magritte generates forms and reports. The Quuve of the Debris Publishing company is a brilliant example of Magritte power: all tables and reports are automatically generated (see http://www.pharo.org/success).

Data validation is also done at the Magritte level instead of being dispersed in the user interface code. This chapter will not cover such aspects. Resources on Magritte are a chapter in the Seaside book (http://book.seaside.st) as well as a tutorial under writing available at https://github.com/SquareBracketAssociates/Magritte.
Figure 9-1  Post management.

Figure 9-2  Administration components.
9.2 Post Description

A description is an object that specifies information on the data of our model as well as its type, whether the information is mandatory, if it should be sorted and what is the default value.

9.2 Post Description

Let us start to describe the five instance variable of TBPost with Magritte. Then we will show how we can get a form generated for us.

We will define the five following methods in the protocol 'magritte-descriptions' of the class TBPost. Note that the method names are not important but we follow a convention. This is the pragma <magritteDescription> (method annotation) that allows Magritte to identify descriptions.

The post title is a string of characters that is mandatory.

```
TBPost >> descriptionTitle
   <magritteDescription>
   ^ MAStringDescription new
      accessor: #title;
      beRequired;
      yourself
```

A post test is a multi-line that is mandatory.

```
TBPost >> descriptionText
   <magritteDescription>
   ^ MAMemoDescription new
      accessor: #text;
      beRequired;
      yourself
```

The category is represented as a string and it does not have to be given. In such case the post will be sorted in the 'Unclassified' category.

```
TBPost >> descriptionCategory
   <magritteDescription>
   ^ MAStringDescription new
      accessor: #category;
      yourself
```

The post creation time is important since it is used to sort posts. It is then required.

```
TBPost >> descriptionDate
   <magritteDescription>
   ^ MADateDescription new
      accessor: #date;
      beRequired;
      yourself
```
The visible instance variable should be a Boolean and it is required.

```small
TBPost >> descriptionVisible
    ^ MABooleanDescription new
    accessor: #visible;
    beRequired;
    yourself
```

We could enrich the descriptions so that it is not possible to publish a post with a date before the current day. We could change the description of a category to make sure that a category is part of a predefined list of categories. We do not do it to keep it to the main point.

9.3 **Automatic Component Creation**

Once a post described we can generate a Seaside component by sending a message `asComponent` to an post instance.

```small
aTBPost asComponent
```

Let us see how we can use this in the following.

9.4 **Building a post report**

We will develop a new component that will be used by the component TBAadminComponent. The TBPostReport component is a report that will contain all the posts. As we will see below the report Seaside component will be generated automatically from Magritte. We could have develop only one component but we prefer to distinguish it from the admin component for future evolution.

**The PostsReport Component**

Post list is displayed using a report dynamically generated by Magritte. We will use Magritte to implement the different behaviors of the admin activity (post list, post creation, edition, delete of a post).

The component TBPostsReport is a subclass of TBSMagritteReport that manages reports with Bootstrap.

```small
TBSMagritteReport subclass: #TBPostsReport
    instanceVariableNames: '
    classVariableNames: '
    package: 'TinyBlog-Components'
```

We add a creation method that takes a blog as argument.
Let us now revise our TBAdminComponent to display this report.

We add an instance variable `report` and its accessors in the class `TBAdminComponent`.

```smalltalk
TBScreenComponent subclass: #TBAdminComponent
    instanceVariableNames: 'report'
    classVariableNames: '
    package: 'TinyBlog-Components'

TBAdminComponent >> report
    ^ report

TBAdminComponent >> report: aReport
    report := aReport

Since the report is a son component of the admin component we should not forget to redefine the method `children`. Note that the collection contains the subcomponents defined in the superclass (header component) and those in current class (report component).

```smalltalk
TBAdminComponent >> children
    ^ super children copyWith: self report

In initialize method we instantiate a report by giving it a blog instance.

```smalltalk
TBAdminComponent >> initialize
    super initialize.
    self report: (TBPostsReport from: self blog)

Let us modify the admin part rendering to display the report.

```smalltalk
TBAdminComponent >> renderContentOn: html
    super renderContentOn: html.

You can test this change by refreshing your web browser.
9.6 Filter Columns

By default, a report displays the full data of each post. However, some columns are not useful. We should filter the columns. Here we only keep the title, category, and publication date.

We add a class method for the column selection and modify the method from:

```smalltalk
TBPostsReport class >> filteredDescriptionsFrom: aBlogPost
"Filter only some descriptions for the report columns."

^ aBlogPost magritteDescription
  select: [ :each | #(title category date) includes: each accessor selector ]
```

```smalltalk
TBPostsReport class >> from: aBlog
| allBlogs |
allBlogs := aBlog allBlogPosts.
^ self rows: allBlogs description: (self filteredDescriptionsFrom: allBlogs anyOne)
```

Figure 9-3 shows the situation that you should get.
9.7 **Report Enhancements**

The previous report is pretty raw. There is no title on columns and the display column order is not fixed. This can change from one instance to the other. To handle this, we modify the description for each instance variable. We specify a priority and a title (message label:) as follows:

```small
TBPost >> descriptionTitle
<magritteDescription>
  ^MAStringDescription new
    label: 'Title';
    priority: 100;
    accessor: #title;
    beRequired;
    yourself

TBPost >> descriptionText
<magritteDescription>
  ^MAMemoDescription new
    label: 'Text';
    priority: 200;
    accessor: #text;
    beRequired;
    yourself

TBPost >> descriptionCategory
<magritteDescription>
  ^MAStringDescription new
    label: 'Category';
    priority: 300;
    accessor: #category;
    yourself

TBPost >> descriptionDate
<magritteDescription>
  ^MADateDescription new
    label: 'Date';
    priority: 400;
    accessor: #date;
    beRequired;
    yourself

TBPost >> descriptionVisible
<magritteDescription>
  ^MABooleanDescription new
    label: 'Visible';
    priority: 500;
    accessor: #visible;
    beRequired;
    yourself
```

You should obtain the situation such as represented by Figure 9-4.
9.8 Post Administration

We can now put in place a CRUD (Create Read Update Delete) allowing to generate posts. For this, we will add a new column (instance of MACommandColumn) to the report. This column will group the different operations using the addCommandOn: message. This method allows one to define a link that will execute a method of the current object. We give access to the blog the report is build for.

```plaintext
TBSMagritteReport subclass: #TBPostsReport
    instanceVariableNames: 'blog'
    classVariableNames: ''
    package: 'TinyBlog-Components'

TBSMagritteReport >> blog
    ^ blog

TBSMagritteReport >> blog: aTBBlog
    blog := aTBBlog

The method from: adds a new column to the report. It groups the different operations.

TBPostsReport class >> from: aBlog
| report blogPosts |
blogPosts := aBlog allBlogPosts.
report blog: aBlog.
report addColumn: (MACommandColumn new
```
9.9 Post Addition

```plaintext
addCommandOn: report selector: #viewPost: text: 'View';
yourself;
addCommandOn: report selector: #editPost: text: 'Edit';
yourself;
addCommandOn: report selector: #deletePost: text: 'Delete';
yourself).
^ report
```

We will have to define the methods linked to each operation in the following section.

In addition this method is a bit lengthy and it does not separate the report definition from the operation definition. A possible solution is to create an instance method named `addCommands` and to call it explicitly. Try to do it to practice.

9.9 Post Addition

Addition a post is not associated with a post and we place just before the main report. Since this behavior is then part of the component `TBPostsReport`, we should redefine the method `renderContentOn:` of the component `TBPostsReport` to insert a link `add`.

```plaintext
TBPostsReport >> renderContentOn: html
    html tbsGlyphIcon iconPencil.
    html anchor
        callback: [ self addPost ];
        with: 'Add post'.
    super renderContentOn: html
```

Login another time and you should get the situation as it is represented in Figure 9-5.

9.10 CRUD Action Implementation

Each action (Create/Read/Update/Delete) should invoke methods of the instance of `TBPostsReport`. We implement them now. A personalized form is built based on the requested operation (it is not necessary to have a save button when the user is just viewing a post).

9.11 Post Addition

Let us begin with post addition. The following method `renderAddPostForm:` illustres the power of Magritte to generate forms:
Here the message asComponent, sent to the object of class TBPost, creates directly a component. We add a decoration to this component to manage ok/cancel.

The method addPost displays the component returned by the method renderAddPostForm: and when a new post is created, it is added for the blog. The method writeBlogPost: saves the changes the user may do.

In this method we see another use of the message call: to give the control to a component. The link to add a post allows one to display a creation form that we will make better looking later (See Figure 9-6).
9.11 Post Addition

Figure 9-6 Basic rendering of a post.

**Post Display**

To display a post in read-only mode, we define two methods similar to the previous. Note that we use the `readonly: true` to indicate that the form is not editable.

```plaintext
TBPostsReport >> renderViewPostForm: aPost
  ^ aPost asComponent
  addDecoration: (TBSMagritteFormDecoration buttons: { #cancel
    -> 'Back' });
  readonly: true;
  yourself

Looking at a post does not require any extra action other than rendering it.

```plaintext
TBPostsReport >> viewPost: aPost
  self call: (self renderViewPostForm: aPost)
```

**Post Edition**

To edit a post, we use the same approach.

```plaintext
TBPostsReport >> renderEditPostForm: aPost
  ^ aPost asComponent addDecoration: (TBSMagritteFormDecoration buttons: { #save
    -> 'Save post'.
    #cancel -> 'Cancel'});
  yourself

Now the method `editPost:` gets the value of the `call:` message and saves the changes made.
```
Administration Web Interface and Automatic Form Generation

Removing a post

We must now adding the method removeBlogPost: to the class TBBlog:

```smalltalk
TBBlog >> removeBlogPost: aPost
    posts remove: aPost ifAbsent: [ ].
    self save.
```

Let us add a unit test:

```smalltalk
TBBlogTest >> testRemoveBlogPost
    self assert: blog size equals: 1.
    blog removeBlogPost: blog allBlogPosts anyOne.
    self assert: blog size equals: 0
```

To avoid an unwanted operation, we use a modal dialog so that the user confirms the deletion of the post. One the post is displayed, the list of managed posts by TBPostsReport is changed and should be refresh.

```smalltalk
TBPostsReport >> deletePost: aPost
    (self confirm: 'Do you want remove this post ?')
    ifTrue: [ blog removeBlogPost: aPost ]
```

9.12 Refreshing Posts

The methods addPost: and deletePost: are working well but the display is not refreshed. We need to refresh the post lists using the expression self refresh.

```smalltalk
TBPostsReport >> refreshReport
    self rows: blog allBlogPosts.
    self refresh.

TBPostsReport >> addPost
    | post |
    post := self call: (self renderAddPostForm: TBPost new).
    post ifNotNil: [ blog writeBlogPost: post.
        self refreshReport ]

TBPostsReport >> deletePost: aPost
    (self confirm: 'Do you want remove this post ?')
    ifTrue: [ blog removeBlogPost: aPost.
        self refreshReport ]
```
The report is not working and it even manage input constraints: for example, mandatory fields should be filled up.

**Better Form Look**

To take advantage of Bootstrap, we will modify Magritte definitions. First we specify that the report rendering based on Bootstrap.

A container in Magritte is the element that will contain the other components created from descriptions.

```smaller
TBPost >> descriptionContainer
  <magritteContainer>
    ^ super descriptionContainer
      componentRenderer: TBSMagritteFormRenderer;
      yourself
  
We want can now pay attention of the different input fields and improve their appearance.

```smaller
TBPost >> descriptionTitle
  <magritteDescription>
    ^ MAMStringDescription new
      label: 'Title';
      priority: 100;
      accessor: #title;
      requiredErrorMessage: 'A blog post must have a title.';
      comment: 'Please enter a title';
      componentClass: TBSMagritteTextInputComponent;
      beRequired;
      yourself

TBPost >> descriptionText
  <magritteDescription>
    ^ MAMMemoDescription new
      label: 'Text';
      priority: 200;
      accessor: #text;
      beRequired;
      requiredErrorMessage: 'A blog post must contain a text.';
      comment: 'Please enter a text';
      componentClass: TBSMagritteTextAreaComponent;
      yourself

TBPost >> descriptionCategory
  <magritteDescription>
    ^ MAMStringDescription new
      label: 'Category';
      priority: 300;
      accessor: #category;
      comment: 'Unclassified if empty';
Based on new Magritte descriptions, forms generated now use Bootstrap. For example, the post form edition should not looks like Figure 9-7.

9.14 Conclusion

In this chapter we defined the administration of TinyBlog based on report built out of the posts contained in the current blog. We added links to manage CRUD for each post. What we show is that adding descriptions on post let us generate Seaside components automatically.
In this chapter we will show you how to deploy your Pharo application. In particular we will show how to deploy on the cloud.

10.1 Deploying in the cloud

Now that TinyBlog is ready we will see how we can deploy your application on a server on the web. If you want to deploy your application on a server that you administrate, we suggest reading the last chapter of "Enterprise Pharo: a Web Perspective" (http://books.pharo.org). In the following we present a simpler solution offered by PharoCloud.

10.2 Login on PharoCloud

PharoCloud is hosting Pharo applications and it offers the possibility to freely tests its services (ephemeric cloud subscription).

Prepare your PharoCloud account:

- Create an account on http://pharocloud.com
- Activate your account
- Connect to this account
- Activate "Ephemeric Cloud" to get an id (API User ID) and password (API Auth Token)
- Click on "Open Cloud Client” and login with the previous ids
• Once connected, you should get a web page that allows you to upload a zip archive containing a Pharo image and its companion Pharo changes file.

10.3 **Preparing your Pharo image Pharo for PharoCloud**

**Get a fresh new image**

You should

• First download a fresh PharoWeb image from http://files.pharo.org/mooc/image/PharoWeb-60.zip.

• Launch this image and now we will configure it.

**Seaside configuration**

We remove the Seaside demo applications and the development tools

```
"Seaside Deployment configuration"
WAAadmin clearAll.
WAAadmin applicationDefaults removeParent: WADevelopmentConfiguration instance.
WAFFileHandler default: WAFFileHandler new.
WAFFileHandler default
  preferenceAt: #fileHandlerListingClass
  put: WAHtmlFileHandlerListing.
WAAadmin defaultDispatcher
  register: WAFFileHandler default
  at: 'files'.
```

**Loading TinyBlog**

We load the latest version of the TinyBlog application. To load the version we propose you can use:

```
"Load TinyBlog"
Gofer new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  package: 'ConfigurationOfTinyBlog';
  load.
#ConfigurationOfTinyBlog asClass loadFinalApp.

"Create Demo posts if needed"
#TBBlog asClass createDemoPosts.
```

You can also load your TinyBlog code from your Smalltalkhub repository. For example doing:
10.4 Manually deploying on PharoCloud’s Ephemeric Cloud

"Load TinyBlog"
Gofer new
    smalltalkhubUser: 'XXXX' project: 'TinyBlog';
    package: 'TinyBlog';
    load.

"Create Demo posts if needed"
#TBBlog asClass createDemoPosts.

TinyBlog as Default Seaside Application

We now set Tinyblog as the default Seaside application and we run the HTTP webserver:
"Tell Seaside to use TinyBlog as default app"
WADispatcher default defaultName: 'TinyBlog'.

"Register TinyBlog on Seaside"
#TBApplicationRootComponent asClass initialize.

Lancer Seaside:
    "Start HTTP server"
    ZnZincServerAdaptor startOn: 8080.

Save the Image

Save your image (Menu World > save) and locally test it in your web browser at: http://localhost:8080.

10.4 Manually deploying on PharoCloud’s Ephemeric Cloud

• Create a zip archive that contains the previously saved images and changes files: PharoWeb.image et PharoWeb.changes.
• Drag and drop this zip file on the Ephemeric Cloud and activate the image using the play button as shown in Figure 10-1.

By clicking on the public URL given by PharoCloud you will be able to display your TinyBlog application as shown by Figure 10-2.

10.5 Automatic Deployment on PharoCloud’s Ephemeric Cloud

Instead of creating a zip archive and using your web browser, the documentation of PharoCloud (http://docs.swarm.pharocloud.com/) shows how to deploy automatically by executing the following code (it takes some time):
Figure 10-1  Ephemeric Cloud administration Pharo image.

Figure 10-2  Your TinyBlog Application on PharoCloud.
### 10.6 About Dependencies

Good development practice in Pharo are to specific explicitly the dependencies on the used packages. This ensures the fact that we can reproduce a software artefact. Such reproducibility supports then the use of an integration server such as Travis or Jenkins. For this a configuration (a special class) defines the architecture of a project (dependencies to other projects as well as structure of your projects) and the associated version. This is this way that we build automatically the PharoWeb image.

In this book we do not cover this point. A full chapter is dedicated to the expression of configurations in the book named "Deep Into Pharo" (cf. [http://books.pharo.org](http://books.pharo.org)).
Part II

Optional Elements
This chapter contains the expressions to load the code described in each of the chapters. Such expressions can be executed in the PharoWeb 5 image (http://files.pharo.org/mooc/image/PharoWeb-5.0.zip) or in PharoWeb 6.1 (http://files.pharo.org/mooc/image/PharoWeb-61.zip).

When you start for example the chapter 4, you can load all the code of the previous chapters (1, 2, and 3) by following the process described in the following section 'Chapter 4'.

Obviously, we believe that this is better that you use your own code but having our code at hand can help you in case you would be stuck.

11.1 **Chapter 3: Extending and Testing the Model**

You can load the correction of the previous chapter as follow:

```smalltalk
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter2solution;
    configuration: 'TinyBlog';
    load.
```

Run the tests! To do so, you can use the TestRunner (World menu > Test Runner), look for the package TinyBlog-Tests and click on "Run Selected". All tests should be green.

11.2 **Chapter 4: Data Persitency using Voyage and Mongo**

You can load the correction of the previous chapter as follow:
11.3 Chapter 5: First Steps with Seaside

You can load the correction of the previous chapter as follow:

```smalltalk
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter3solution;
    configuration: 'TinyBlog';
    load.
```

Once loaded execute the tests.

To test the application, start the HTTP server:

```smalltalk
ZnZincServerAdaptor startOn: 8080.
```

Open your web browser at http://localhost:8080/TinyBlog

You may need to recreate some posts as follows:

```smalltalk
TBBlog reset ; createDemoPosts
```

11.4 Chapitre 6: Web Components for TinyBlog

You can load the correction of the previous chapter as follow:

```smalltalk
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter4solution;
    configuration: 'TinyBlog';
    load.
```

Same process as above.

11.5 Chapitre 7: Managing Categories

You can load the correction of the previous chapter as follow:

```smalltalk
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter5solution;
    configuration: 'TinyBlog';
    load.
```
11.6 Chapitre 8: Authentication and Session

To test the application, start the HTTP server:

```
ZnZincServerAdaptor startOn: 8080.
```

Open your web browser at http://localhost:8080/TinyBlog

You may need to recreate some posts as follows:

```
TBBlog reset ; createDemoPosts
```

11.6 Chapitre 8: Authentication and Session

You can load the correction of the previous chapter as follow:

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter7solution;
    configuration: 'TinyBlog';
    load
```

To test the application, start the HTTP server:

```
ZnZincServerAdaptor startOn: 8080.
```

11.7 Chapitre 9: Administration Web Interface and Automatic Form Generation

You can load the correction of the previous chapter as follow:

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter8solution;
    configuration: 'TinyBlog';
    load
```

11.8 Chapitre 10: Deploying TinyBlog

You can load the correction of the previous chapter as follow:

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #chapter9solution;
    configuration: 'TinyBlog';
    load
```

You should create an account on PharoCloud and use your IDs in the following code:
ephUser := '<your PharoCloud Ephemeric login>'.
ephToken := '<your PharoCloud Ephemeric passwod>'.

"Seaside Deployment configuration"
WAAdmin clearAll.
WAAdmin applicationDefaults removeParent: WADevelopmentConfiguration instance.
WAFileHandler default: WAFileHandler new.
WAFileHandler default
  preferenceAt: #fileHandlerListingClass
  put: WAHtmlFileHandlerListing.
WAAdmin defaultDispatcher
  register: WAFileHandler default
  at: 'files'.

"Load TinyBlog"
Gofer new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  package: 'ConfigurationOfTinyBlog';
  load.
#ConfigurationOfTinyBlog asClass loadFinalApp.

"Create Demo posts if needed"
#TBBlog asClass createDemoPosts.

"Tell Seaside to use TinyBlog as default app"
WADispatcher default defaultName: 'TinyBlog'.

"Register TinyBlog on Seaside"
TBApplicationRootComponent initialize.

Metacello new
  smalltalkhubUser: 'mikefilonov' project: 'EphemericCloudAPI';
  configuration: 'EphemericCloudAPI';
  load.

"deployment on PharoCloud"
client := #EphemericCloudClient asClass userID: ephUser authToken: ephToken.
(client publishSelfAs: 'TinyBlog')
  ifTrue:[ ZnZincServerAdaptor startOn: 8080 ]
  ifFalse: [ client lastPublishedInstance hostname ]

When you execute the following code (it will take some time), your Pharo image will be automatically be deployed on PharoCloud.
CHAPTER 12

Save your code

When you save the Pharo image (left click on the Pharo background to access the World menu and select 'save' menuentry), it contains all objects of the system as well as all classes. This solution is useful but fragile. We will show you how Pharoers save their code as packages and send them on a remote dedicated server. Until Pharo 6.0, Pharo offers the possibility to save packages using Monticello: the package versioning tool of Pharo on http://smalltalkhub.com or http://ss3.gemstone.com. From Pharo 70 Pharo still support Monticello but offers a much better way to store your code on github, bitbucket, or gitlab using git with a new dedicated tool.

12.1 Up to Pharo 6.0

The Pharo Mooc (http://mooc.pharo.org) contains video that describes the procedure to save code such as the video of the Week 1 that shows how to develop a counter application and save the code: http://rmod-pharo-mooc.lille.inria.fr/MOOC/Videos/W1/C019-W1S-Videos-Redo-Counter-Traditional-FR-V3-HD_720p_4Mbs.m4v.

Create a code repository

There are multiple online servers to freely host your code repositories such as Smalltalkhub http://smalltalkhub.com or SS3 http://ss3.gemstone.com.

- Create an account on http://smalltalkhub.com/.
• Create a project named "TinyBlog" (if you encounter connection problems because this web site is still in beta version, try with another web browser or use http://ss3.gemstone.com).

Save your packages

• In Pharo, open Monticello Browser using the world menu (left click on the Pharo background).

• Add a new repository of type SmalltalkHub or HTTP for http://ss3.gemstone.com.

• Select this repository and select the 'Add to package...' in its contextual menu (right click) to add this repository to the TinyBlog package.

• You can now select you 'TinyBlog' package and click on the 'Save' button.

• Write a description for your commit and save. Your code has been sent to the server.

The code of your TinyBlog application is now saved in your 'TinyBlog' repository on remote server (e.g. Smalltalkhub). You can now load this code in a new Pharo image. In this tutorial, we recommend you to always use a Pharo image with all Web packages already loaded (Pharo Web image) as mentioned in the first chapter. This way, you can always load your code without dealing with dependencies with third-party packages.

12.2 With Pharo 70

We suggest you read the chapter in the book ”Managing Your Code with Iceberg” (available at http://books.pharo.org).

We list the key points here:

• Create a project on http://www.github.com or similar.

• Use Iceberg pour add a un projet, choose to clone from github by example.

• Create a folder 'src' with the filelist or using the command line in the folder that you just cloned.

• Open your project and add your packages (Define a baseline to be able to reload your code – check https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Baselines.md)

• Commit your code.

• Push your code on github.