

TinyBlog: Créer votre Première Web App avec Pharo

Olivier Auverlot, Stéphane Ducasse et Luc Fabresse

September 18, 2018
master @ c5d975d*

Copyright 2017 by Olivier Auverlot, Stéphane Ducasse et Luc Fabresse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:

<http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Contents

Illustrations	v
1 A propos de ce livre	1
1.1 Structure	1
1.2 Installation de Pharo	2
1.3 Règles de nommage	3
1.4 Ressources	3
I Tutoriel de base	
2 Modèle de l'application TinyBlog	7
2.1 La classe TBPPost	7
2.2 Gérer la visibilité d'un post	8
2.3 Initialisation	9
2.4 Méthodes de création	9
2.5 Création de posts	10
2.6 Ajout de quelques tests unitaires	10
2.7 Interrogation d'un post	11
2.8 Conclusion	12
3 TinyBlog : extension du modèle et tests unitaires	13
3.1 La classe TBBlog	13
3.2 Un seul blog	14
3.3 Tester les règles métiers	15
3.4 Un premier test	16
3.5 Améliorons la couverture de test	16
3.6 Autres fonctionnalités	16
3.7 Données de test	18
3.8 Futures évolutions	19
3.9 Conclusion	19

4	Persistence des données de TinyBlog avec Voyage et Mongo	21
4.1	Configurer Voyage pour sauvegarder des objets TBBlog	22
4.2	Sauvegarde d'un blog	23
4.3	Revision des tests	23
4.4	Utilisation de la base	24
4.5	Si nous devons sauvegarder les posts [Discussion]	25
4.6	Deployer avec une base Mongo [Optionnel]	26
4.7	Conclusion	28
5	Commencer avec Seaside	29
5.1	Démarrer Seaside	30
5.2	Bootstrap for Seaside	30
5.3	Point d'entrée de l'application	31
5.4	Premier rendu simple	33
5.5	Architecture	34
5.6	Conclusion	36
6	Des composants web pour TinyBlog	37
6.1	Composants visuels	38
6.2	Utilisation du composant Screen	39
6.3	Pattern de définition de composants	40
6.4	Ajouter quelques bulletins au blog	40
6.5	Définition du composant TBHeaderComponent	41
6.6	Utilisation du composant header	41
6.7	Relation composite-composant	42
6.8	Rendu visuel de la barre de navigation	42
6.9	Liste des posts	43
6.10	Le composant Post	45
6.11	Afficher les bulletins (posts)	46
6.12	Débugger les erreurs	46
6.13	Affichage de la liste des posts avec Bootstrap	47
6.14	Cas d'instanciation de composants dans renderContentOn:	47
6.15	Conclusion	48
7	Gestion des catégories	49
7.1	Affichage des bulletins par catégorie	49
7.2	Rendu des catégories	51
7.3	Mise à jour de la liste des bulletins	52
7.4	Look et agencement	53
7.5	Modulariser son code avec des petites méthodes	55
7.6	Conclusion	56
8	Authentification et Session	59
8.1	Composant d'administration simple (v1)	60
8.2	Ajout d'un bouton 'admin'	60
8.3	Revisons la barre de navigation	61

8.4	Activation du bouton d'admin	62
8.5	Ajout d'un bouton 'disconnect'	63
8.6	Composant fenêtre modale d'identification	64
8.7	Rendu du composant d'identification	66
8.8	Intégration du composant d'identification	67
8.9	Gestion naive des logins	68
8.10	Gestion des erreurs	68
8.11	Modélisation des administrateurs	69
8.12	Administrateur pour un blog	71
8.13	Définir un administrateur	71
8.14	Intégration du compte administrateur	72
8.15	Stocker l'administrateur courant en session	72
8.16	Définition et utilisation d'une classe session spécifique	73
8.17	Stockage de l'administrateur courant en session	74
8.18	Navigation simplifiée vers la partie administration	74
8.19	Déconnexion	75
8.20	Navigation simplifiée vers la partie publique	75
8.21	Conclusion	76
9	Interface web d'administration et génération automatique	77
9.1	Décrire les données métiers avec Magritte	77
9.2	Description d'un bulletin	79
9.3	Création automatique de composant	80
9.4	Mise en place d'un rapport des bulletins	80
9.5	Intégration de PostsReport dans AdminComponent	81
9.6	Filtrer les colonnes	82
9.7	Amélioration du rapport	82
9.8	Administration des bulletins	84
9.9	Gérer l'ajout d'un bulletin	85
9.10	Implémentation des actions CRUD	86
9.11	Ajouter un bulletin	86
9.12	Gérer le rafraîchissement des données	88
9.13	Amélioration de l'apparence des formulaires	89
9.14	Conclusion	91
10	Déploiement de TinyBlog	93
10.1	Déployer dans le cloud	93
10.2	Hébergement sur PharoCloud	93
10.3	Préparation de l'image Pharo pour PharoCloud	94
10.4	Déploiement manuel sur Ephemer Cloud de PharoCloud	95
10.5	Déploiement automatique sur Ephemer Cloud de PharoCloud	97
10.6	A propos de dépendances	97

II Éléments optionnels

11	Charger le code des chapitres	101
11.1	Chapitre 3 : Extension du modèle et tests unitaires	101
11.2	Chapitre 4 : Persistance des données de TinyBlog avec Voyage et Mongo . .	102
11.3	Chapitre 5 : Commencer avec Seaside	102
11.4	Chapitre 6 : Des composants web pour TinyBlog	102
11.5	Chapitre 7 : Gestion des catégories	103
11.6	Chapitre 8 : Authentification et Session	103
11.7	Chapitre 9 : Interface Web d'administration et génération automatique . . .	103
11.8	Chapitre 10 : Déploiement de TinyBlog	104
12	Sauver votre code	107

Illustrations

1-1	L'application TinyBlog.	2
2-1	TBPost une classe très simple gérant principalement des données.	7
2-2	Inspecteur sur une instance de TBPost.	10
3-1	TBBlog une classe très simple.	14
5-1	Lancer le serveur.	29
5-2	Vérification que Seaside fonctionne.	30
5-3	Accès à la bibliothèque Bootstrap.	31
5-4	Comprendre un élément et son code en Bootstrap.	32
5-5	TinyBlog est bien enregistrée.	33
5-6	Une page quasi vide mais servie par Seaside.	33
5-7	Les composants composant l'application TinyBlog (en mode non admin). . .	35
5-8	Architecture de TinyBlog.	35
6-1	L'architecture des composants utilisateurs (par opposition à administration).	37
6-2	Les composants visuels de l'application TinyBlog.	38
6-3	Le composant <code>ApplicationRootComponent</code> utilise de manière temporaire le composant <code>ScreenComponent</code> qui a un <code>HeaderComponent</code> . .	39
6-4	Premier visuel du composant <code>TBScreenComponent</code>	40
6-5	TinyBlog avec une barre de navigation.	43
6-6	Le composant <code>ApplicationRootComponent</code> utilise le composant <code>PostsListComponent</code>	44
6-7	TinyBlog avec une liste de bulletins plutôt élémentaire.	44
6-8	Ajout du composant <code>Post</code>	45
6-9	TinyBlog avec une liste de posts.	47
7-1	L'architecture des composants de la partie publique avec catégories.	49
7-2	Catégories afin de sélectionner les posts.	53
7-3	Liste des catégories avec un meilleur agencement.	54
7-4	TinyBlog UI version finale.	56

8-1	Gérant l'authentification pour accéder à l'administration.	59
8-2	Lien simple vers la partie administration.	60
8-3	Barre de navigation avec un bouton admin.	61
8-4	Affichage du composant admin en cours de définition.	63
8-5	Aperçu du composant d'identification.	65
8-6	Message d'erreur en cas d'identifiants erronés.	70
8-7	Navigation et identification dans TinyBlog.	73
9-1	Gestion des bulletins.	78
9-2	Composants pour l'administration.	78
9-3	Rapport Magritte contenant les bulletins du blog.	83
9-4	Administration avec un rapport.	84
9-5	Rapport des bulletins avec des liens d'édition.	86
9-6	Affichage rudimentaire d'un bulletin.	87
9-7	Formulaire d'ajout d'un post avec Bootstrap.	90
10-1	Administration des images Pharo sur Ephemer Cloud.	96
10-2	Votre application TinyBlog sur PharoCloud.	96



A propos de ce livre

Tout au long de ce projet, nous allons vous guider pour développer et enrichir une application Web, nommée TinyBlog, pour gérer un ou plusieurs blogs. La figure 1-1 montre l'état final de l'application. L'idée est qu'un visiteur du site Web puisse voir les posts et que l'auteur du blog puisse se connecter sur le site pour administrer le blog c'est-à-dire ajouter, supprimer ou modifier des posts.

Tinblog est une petite application pédagogique qui va vous montrer comment développer et déployer une application web en utilisant Pharo / Seaside / Mongo et d'autres frameworks tels que NeoJSON.

Notre idée est que par la suite vous pourrez réutiliser cette infrastructure pour créer vos propres applications Web.

1.1 Structure

Dans la première partie appelée "Tutoriel de base", vous allez développer et déployer, TinyBlog, une application et son administration en utilisant Pharo et le framework Seaside ainsi que d'autres bibliothèques comme Voyage ou Magritte. Le déploiement en utilisant la base de données Mongo est optionnel mais cela vous permet de voir que Voyage est une façade élégante et simple pour faire persister des données notamment dans Mongo.

Dans une seconde partie optionnelle, nous abordons des aspects optionnels tel que l'export de données, l'utilisation de template comme Mustache ou comment exposer votre application via une API REST.

Les solutions proposées dans ce tutoriel sont parfois non optimales afin de vous faire réagir et que vous puissiez proposer d'autres solutions et des améliorations. Notre objectif n'est pas d'être exhaustif. Nous montrons une façon

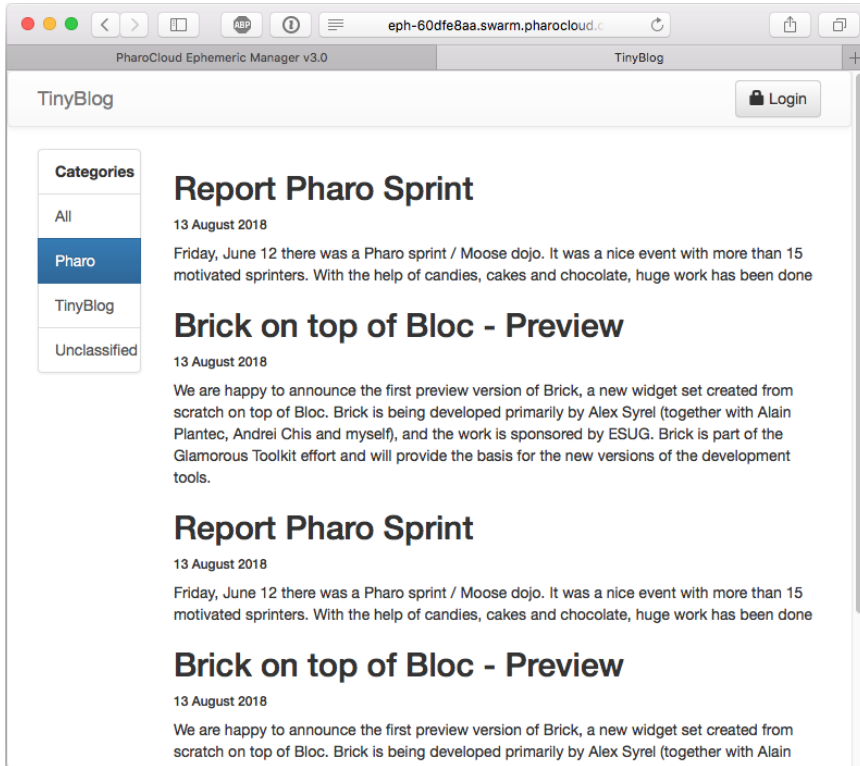


Figure 1-1 L'application TinyBlog.

de faire cependant nous invitons le lecteur à lire les références sur les autres chapitres, livres et tutoriaux Pharo afin d'approfondir son expertise et enrichir son application.

Finalement, afin de vous permettre de ne pas abandonner si vous ne trouvez pas une erreur, le dernier chapitre vous permet de charger le code décrit dans chacun des chapitres.

1.2 Installation de Pharo

Dans ce tutoriel, nous supposons que vous utilisez Pharo 6.1 avec une image dans laquelle ont été chargés des bibliothèques et des frameworks spécifiques pour le développement d'applications Web: Seaside (le serveur d'application web à base de composants), Magritte (un framework de description pour la génération automatique de rapport), Bootstrap (la bibliothèque de rendu web), Voyage (un framework pour sauver vos objets) et quelques autres.

Pour développer votre application TinyBlog et suivre ce tutoriel, nous vous suggérons de toujours utiliser l'image disponible à l'adresse suivante: <http://mooc.pharo.org/image/PharoWeb-61.zip> car elle contient tous les packages nécessaires.

Vous pouvez construire l'image que nous utilisons à l'aide de ce script à exécuter dans une image Pharo 6.1 (<http://pharo.org/download>):

```
Metacello new
  smalltalkhubUser: 'PharoExtras' project: 'PharoWeb';
  configuration: 'PharoWeb';
  version: #stable;
  load
```

1.3 Règles de nommage

Dans la suite, nous préfixons tous les noms de classe par TB (pour TinyBlog). Vous pouvez:

- soit choisir un autre préfixe (par exemple TBM) afin de pouvoir ensuite charger la correction dans la même image Pharo et la comparer à votre propre implémentation,
- soit choisir le même préfixe afin de pouvoir fusionner les solutions proposées avec votre code. L'outil de merge vous montrera les différences et vous permettra d'apprendre des changements. Cette solution est toutefois plus contraignante si vous implémentez des fonctionnalités supplémentaires par rapport aux corrections ou même différemment ce qui est fort probable.

1.4 Ressources

Pharo possède de bonnes ressources pédagogique ainsi qu'une communauté d'utilisateurs accueillante. Voici quelques informations qui peuvent vous être utiles.

- <http://books.pharo.org> contient des ouvrages autour de Pharo. Pharo by Example peut vous aider dans les aspects de découverte du langage et des bibliothèques de base. Entreprise Pharo: a Web Perspective présente d'autres aspects utiles pour le développement web.C
- <http://book.seaside.st> est un des ouvrages sur Seaside. Il est en cours de migration en livre open-source sur <https://github.com/SquareBracketAssociates/DynamicWebDevelopmentWithSeaside>.
- <http://mooc.pharo.org> propose un excellent Mooc (cours en ligne) comprenant plus de 90 vidéos expliquant des points de syntaxes mais aussi de conception objet.

- <http://discord.gg/Sj2rhxn> est le channel discord où nombre de Pharoers échangent et s'entraident.

Part I

Tutoriel de base

Modèle de l'application TinyBlog

Dans ce chapitre, nous développons une partie du modèle de l'application Tinyblog. Le modèle est particulièrement simple : il définit un bulletin. Dans le chapitre suivant nous définissons un blog qui contient une liste de bulletins.

2.1 La classe TBPPost

Nous commençons ici par la représentation d'un bulletin (post) avec la classe TBPPost. Elle est très simple (comme le montre la figure 2-1) et elle définit ainsi:

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

Nous utilisons cinq variables d'instance pour décrire un bulletin sur le blog.

Post
visible
date
title
text
category
isVisible
isUnclassified

Figure 2-1 TBPPost une classe très simple gérant principalement des données.

Variable	Signification
title	Titre du bulletin
text	Texte du bulletin
date	Date de rédaction
category	Rubrique contenant le bulletin
visible	Post visible ou pas ?

Cette classe est également dotée de méthodes d'accès (aussi appelées accesseurs) à ces variables d'instances dans le protocole 'accessing'. Vous pouvez utiliser un refactoring pour créer automatiquement toutes les méthodes suivantes:

```
[ TBPPost >> title
  ^ title
[ TBPPost >> title: aString
  title := aString
[ TBPPost >> text
  ^ text
[ TBPPost >> text: aString
  text := aString
[ TBPPost >> date
  ^ date
[ TBPPost >> date: aDate
  date := aDate
[ TBPPost >> visible
  ^ visible
[ TBPPost >> visible: aBoolean
  visible := aBoolean
[ TBPPost >> category
  ^ category
[ TBPPost >> category: anObject
  category := anObject
```

2.2 Gérer la visibilité d'un post

Ajoutons dans le protocole 'action' des méthodes pour indiquer qu'un post est visible ou pas.

```
[ TBPPost >> beVisible
  self visible: true
[ TBPPost >> notVisible
  self visible: false
```


2.3 Initialisation

La méthode `initialize` (protocole 'initialization') fixe la date à celle du jour et la visibilité à faux. L'utilisateur devra par la suite activer la visibilité. Cela permet de rédiger des brouillons et de ne publier un bulletin que lorsque celui-ci est terminé. Un bulletin est également rangé par défaut dans la catégorie 'Unclassified' que l'on définit au niveau classe. La méthode `unclassifiedTag` renvoie une valeur indiquant que le post n'est pas rangé dans une catégorie.

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

Attention la méthode `unclassifiedTag` est définie au niveau de la classe (cliquer le bouton 'Class' pour la définir). Les autres méthodes sont des méthodes d'instances c'est-à-dire qu'elles seront exécutées sur des instances de la classe `TBPost`.

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

Dans la solution proposée ci-dessus pour la méthode `initialize`, il serait préférable de ne pas faire une référence en dur à la classe `TBPost`. Proposer une solution. La séquence 3 de la semaine 6 du MOOC peut vous aider à mieux comprendre pourquoi (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_67.html) il faut éviter de référencer des classes directement et comment faire.

2.4 Méthodes de création

Coté classe, on définit des méthodes de classe (i.e., exécuter sur des classes) pour faciliter la création de post appartenant ou pas à une catégorie - de telles méthodes sont souvent groupées dans le protocole 'instance creation'.

Nous définissons deux méthodes.

```
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
```

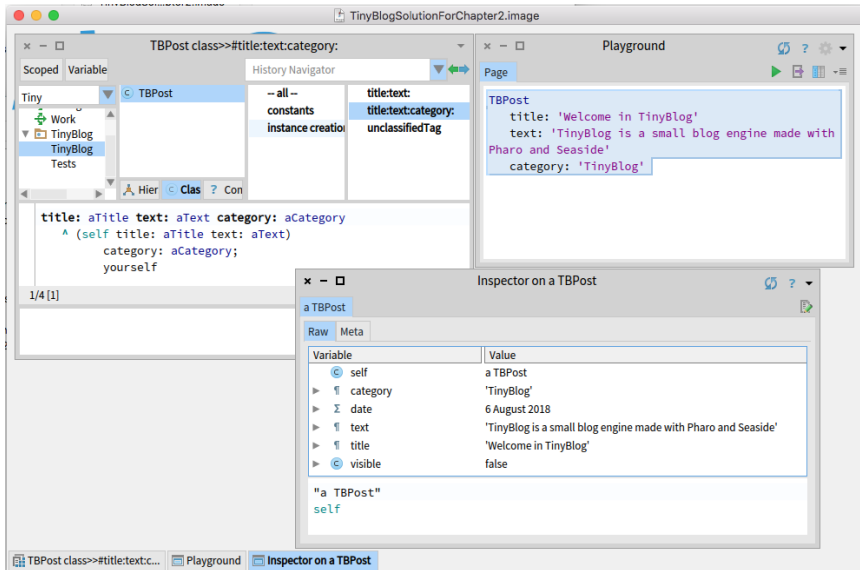


Figure 2-2 Inspecteur sur une instance de TBPPost.

2.5 Création de posts

Créons des posts pour s'assurer que tout fonctionne. Ouvrez l'outil Playground et exécutez l'expression suivante :

```

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

```

Si vous inspectez le code ci-dessus (clic droit sur l'expression et "Inspect it"), vous allez obtenir un inspecteur sur l'objet post nouvellement créé comme représenté sur la figure 2-2.

2.6 Ajout de quelques tests unitaires

Inspecter manuellement des objets n'est pas une manière systématique de vérifier que ces objets ont les propriétés attendues. Bien que le modèle soit simple nous pouvons définir quelques tests. En mode Test Driven Development nous écrivons les tests en premier. Ici nous avons préféré vous laisser définir une petite classe pour vous familiariser avec l'IDE. Mais maintenant nous réparons ce manque.

Nous définissons la classe `TBPPostTest` (comme sous-classe de `TestCase`).

```
TestCase subclass: #TBPostTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'TinyBlog-Tests'
```

Nous définissons deux tests.

```
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.' .
```

Vos tests doivent passer.

2.7 Interrogation d'un post

Dans le protocole 'testing', définissez les deux méthodes suivantes qui permettent respectivement, de demander à un post s'il est visible, et s'il est classé dans une catégorie.

```
TBPost >> isVisible
^ self visible

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

De même il serait préférable de ne pas faire une référence en dur à la classe TBPost dans le corps d'une méthode. Proposer une solution!

De plus, prenons le temps de mettre à jour notre test pour couvrir ce nouvel aspect. Nous simplifions de cette manière la logique de notre test.

```
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

Nous avons développé une première partie du modèle (la classe TBPPost) et défini quelques tests. Nous vous suggérons fortement d'écrire d'autres tests unitaires pour vérifier que ce modèle fonctionne correctement même s'il est simple.



TinyBlog : extension du modèle et tests unitaires

Dans ce chapitre nous étendons le modèle et ajoutons des tests. Notez qu'un bon développeur de méthodologies agiles tel que Test-Driven Development aurait commencé par écrire des tests. En plus, avec Pharo, nous aurions aussi codé dans le débogueur pour être encore plus productif. Nous ne l'avons pas fait car le modèle est simpliste et expliquer comment coder dans le débogueur demande plus de description textuelle. Vous pouvez voir cette pratique dans la vidéo du Mooc intitulée *Coding a Counter in the Debugger* (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_26.html) et lire le livre *Learning Object-Oriented Programming, Design with TDD in Pharo* (<http://books.pharo.org>).

Avant de commencer, reprenez votre code ou reportez-vous au dernier chapitre du livre pour charger le code du chapitre précédent.

3.1 La classe TBBlog

Nous allons développer la classe TBBlog qui contient des bulletins (posts), en écrivant des tests puis en les implémentant (voir la figure 3-1).

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

Nous initialisons la variable d'instance `posts` avec une collection vide.

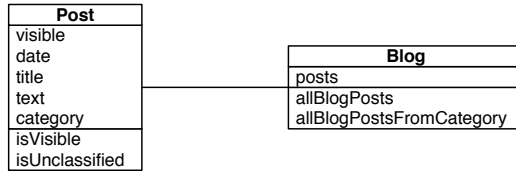


Figure 3-1 TBBlog une classe très simple.

```

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

3.2 Un seul blog

Dans un premier temps nous supposons que nous allons gérer qu'un seul blog. Dans le futur, vous pourrez ajouter la possibilité de gérer plusieurs blogs comme un par utilisateur de notre application. Pour l'instant, nous utilisons donc un singleton pour la classe TBBlog. Faites attention car le schéma de conception Singleton est rarement bien utilisé et peut rendre votre conception rapidement de mauvaise qualité. En effet, un singleton est souvent une sorte de variable globale et rend votre conception moins modulaire. Evitez de faire des références explicites au Singleton dans votre code. Quand vous utilisez un Singleton le mieux est d'y accéder via une variable d'instance qui pourra dans un second temps faire référence à un autre objet sans vous forcer à tout réécrire. Donc ne généralisez pas ce que nous faisons ici.

Comme la gestion du singleton est un comportement de classe, ces méthodes sont définies sur le coté class de la classe TBBlog. Nous définissons une variable d'instance au niveau classe:

```

TBBlog class
  instanceVariableNames: 'uniqueInstance'
  
```

Nous définissons deux méthodes pour gérer le singleton.

```

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

```

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

Nous redéfinissons la méthode de classe initialize afin que la classe soit réinitialisée quand elle est chargée en mémoire.

```

TBBlog class >> initialize
  self reset
  
```

3.3 Tester les règles métiers

Nous allons écrire des tests pour les règles métiers et ceci en mode TDD (Test-Driven Development) c'est-à-dire en développant les tests en premier puis en définissant les fonctionnalités jusqu'à ce que les tests passent.

Les tests unitaires sont regroupés dans une étiquette (tag) `TinyBlog-Tests` qui contient la classe `TBBlogTest` (voir menu item "Add Tag..."). Un tag est juste une étiquette qui permet de trier et grouper les classes à l'intérieur d'un package. Nous utilisons un tag ici pour ne pas avoir à gérer deux packages différents mais dans un projet réel vous définirions un (ou plusieurs) package séparé pour les tests.

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

La méthode `setUp` permet d'initialiser le contexte des tests (aussi appelé *fixture*). Elle est donc exécutée avant chaque test unitaire. Dans cet exemple, elle efface le contenu du blog, lui ajoute un post et en crée un autre qui n'est provisoirement pas enregistré. Faites attention car nous devrons changer cette logique car dans le future a chaque fois que vous exécuterez des tests, vous perdrez votre domaine. C'est un exemple de la sorte d'effets pernicieux qu'un Singleton introduit dans un système.

```
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
```

Afin de tester différentes configurations, les posts `post` et `first` n'appartiennent pas à la même catégorie, l'un est visible et l'autre pas.

Définissons également la méthode `tearDown` qui est exécutée après chaque test et remet le blog à zéro.

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

L'utilisation d'un Singleton montre une de ses limites puisque si vous déployez un blog puis exécutez les tests vous perdrez les posts que vous avez créés car nous les remettons à zéro.

Nous allons développer les tests d'abord puis les fonctionnalités testées. Les fonctionnalités métiers seront regroupées dans le protocole 'action' de la classe TBBlog.

3.4 Un premier test

Commençons par écrire un premier test qui ajoute un post et vérifie qu'il est effectivement ajouté au blog.

```
TBBlogTest >> testAddBlogPost
  blog writeBlogPost: post.
  self assert: blog size equals: 2
```

Ce test ne passe pas (n'est pas vert) car nous n'avons pas défini les méthodes: writeBlogPost:, removeAllPosts et size. Ajoutons-les.

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

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

TBBlog >> size
  ^ posts size
```

Le test précédent doit maintenant passer.

3.5 Améliorons la couverture de test

Ecrivons d'autres tests pour couvrir les fonctionnalités que nous venons de développer.

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

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

3.6 Autres fonctionnalités

Nous allons procéder en suivant une méthodologie dirigée par les tests (Test Driven Development). Nous définissons un test, vérifions que le test ne passe pas. Puis nous définissons la méthode qui était ainsi spécifiée et nous vérifions que le test passe.

Obtenir l'ensemble des posts (visibles et invisibles)

Ajoutons un nouveau test qui échoue :

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

Et le code métier qui permet de le faire passer:

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

Votre nouveau test doit passer.

Obtenir tous les posts visibles

Ajoutons un nouveau test qui échoue :

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

Voici le nouveau code métier ajouté :

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

Votre nouveau test doit passer.

Obtenir tous les posts d'une catégorie

Ajoutons un nouveau test qui échoue :

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

Voici le nouveau code métier ajouté :

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

Votre nouveau test doit passer.

Obtenir tous les posts visibles d'une catégorie

Ajoutons un nouveau test qui échoue :

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

```

self assert: (blog allVisibleBlogPostsFromCategory: 'Second
Category') size equals: 1

```

Voici le nouveau code métier ajouté :

```

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

```

Votre nouveau test doit passer.

Vérifier la gestion des posts non classés

Nous ajoutons un nouveau test pour vérifier que notre fixture ne contient pas de tests non classifiés.

```

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

```

Obtenir la liste des catégories

Ajoutons un nouveau test qui retourne la liste des catégories et qui échoue :

```

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

```

Voici le code métier :

```

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

```

Votre nouveau test doit passer.

3.7 Données de test

Afin de nous aider à tester l'application nous définissons une méthode qui ajoute des posts au blog courant.

```

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: ((TBPst 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);  
writeBlogPost: ((TBPst title: 'The sad story of unclassified  
blog posts' text: 'So sad that I can read this.') visible: true);  
writeBlogPost: ((TBPst title: 'Working with Pharo on the  
Raspberry Pi' text: 'Hardware is getting cheaper and many new  
small devices like the famous Raspberry Pi provide new  
computation power that was one once only available on regular  
desktop computers.' category: 'Pharo') visible: true)
```

Vous pouvez inspecter le résultat de l'évaluation du code suivant :

```
[ TBBlog createDemoPosts ; current
```

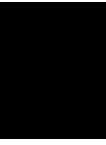
Attention, si vous exécutez plus d'une fois la méthode `createDemoPosts`, le blog contiendra plusieurs exemplaires de ces posts.

3.8 Futures évolutions

Plusieurs évolutions peuvent être apportées telles que: obtenir uniquement la liste des catégories contenant au moins un post visible, effacer une catégorie et les posts qu'elle contient, renommer une catégorie, déplacer un post d'une catégorie à une autre, rendre visible ou invisible une catégorie et son contenu, etc. Nous vous encourageons développer ces fonctionnalités ou de nouvelles que vous auriez imaginé.

3.9 Conclusion

Vous devez avoir le modèle complet de TinyBlog ainsi que des tests unitaires associés. Vous êtes maintenant prêt pour des fonctionnalités plus avancées comme le stockage ou un premier serveur HTTP. C'est aussi un bon moment pour sauver votre code dans votre dépôt en ligne.



Persistance des données de TinyBlog avec Voyage et Mongo

Avoir un modèle d'objets en mémoire fonctionne bien, et sauvegarder l'image Pharo sauve aussi ces objets. Toutefois, il est préférable de pouvoir sauver les objets (les posts) dans une base de données extérieure. Pharo offre plusieurs sérialiseurs d'objets (Fuel en format binaire et STON en format texte). Ces sérialiseurs d'objets sont très puissants et pratiques. Souvent sauver un graphe complet d'objets est réalisé en une seule ligne de code comme explique dans le livre Enterprise Pharo disponible à <http://books.pharo.org>.

Dans ce chapitre, nous voulons vous présenter une autre option : la sauvegarde dans une base de donnée orientée documents telle que Mongo (<https://www.mongodb.com>) en utilisant le framework Voyage. Voyage est un framework qui propose une API unifiée permettant d'accéder à différentes bases de données documents comme Mongo ou UnQLite afin d'y stocker des objets.

Dans ce chapitre, nous allons commencer par utiliser la capacité de Voyage à simuler une base extérieure. Ceci est très pratique en phase de développement. Dans un second temps, nous installerons une base de données Mongo et nous y accéderons à travers Voyage.

Comme pour chacun des chapitres précédents vous pouvez charger le code comme indiqué dans le dernier chapitre.

4.1 Configurer Voyage pour sauvegarder des objets TB-Blog

Grâce à la méthode de classe `isVoyageRoot`, nous déclarons que les objets de la classe `TBBlog` doivent être sauvés dans la base en tant qu'objets racines. Cela veut dire que nous aurons autant de documents que d'objets instance de cette classe.

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

Nous devons ensuite soit créer une connexion sur une base de données réelle soit travailler en mémoire. C'est cette dernière option que nous choisissons pour l'instant en utilisant cette expression.

```
VOMemoryRepository new enableSingleton.
```

Le message `enableSingleton` indique à Voyage que nous n'utilisons qu'une seule base de donnée.

Nous définissons une méthode `initializeVoyageOnMemoryDB` dont le rôle est d'initialiser correctement la base.

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

Nous définissons la méthode de classe `reset` afin de réinitialiser la base de données. Nous redéfinissons également la méthode class `initialize` pour réinitialiser la base de données lorsque l'on charge le code c'est-à-dire lorsque la classe `TBBlog` est chargée.

```
TBBlog class >> reset
  self initializeVoyageOnMemoryDB
```

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

N'oubliez pas d'exécuter la méthode `initialize` une fois la méthode définie en exécutant l'expression `TBBlog initialize`.

Le cas de la méthode `current` est plus délicat. Avant l'utilisation de Mongo, nous avions un singleton tout simple. Cependant utiliser un Singleton ne fonctionne plus car imaginons que nous ayons sauvé notre blog et que le serveur s'éteigne par accident ou que nous rechargeons une nouvelle version du code. Ceci conduirait à une réinitialisation et création d'une nouvelle instance. Nous pouvons donc nous retrouver avec une instance différente de celle sauvée.

Nous redéfinissons `current` de manière à faire une requête dans la base. Comme pour le moment nous ne gérons qu'un blog il nous suffit de faire

`self selectOne: [:each | true]` ou `self selectAll anyOne`. Nous nous assurons de créer une nouvelle instance et la sauvegarder si aucune instance n'existe dans la base.

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

La variable `uniqueInstance` qui servait auparavant à stocker le singleton `TBBlog` peut être enlevée.

```
TBBlog class
    instanceVariableNames: ''
```

4.2 Sauvegarde d'un blog

Nous devons maintenant modifier la méthode `writeBlogPost:` pour sauver le blog lors de l'ajout d'un post.

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

Nous pouvons aussi modifier la méthode `remove` afin de sauver le nouvel état d'un blog.

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

4.3 Revision des tests

Maintenant que nous sauvons les blogs dans une base (quelle soit en mémoire ou dans une base Mongo), nous devons faire attention car si un test modifie la base, notre base courante (hors test) sera elle aussi modifiée : Cette situation est clairement dangereuse. Un test ne doit pas modifier l'état du système.

Pour résoudre ce problème, avant de lancer un test nous allons garder une référence au blog courant, créer un nouveau contexte puis nous allons utiliser cette référence pour restaurer le blog courant après l'exécution d'un test.

Nous ajoutons la variable d'instance `previousRepository` à la classe `TB-BLogTest`.

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

Ensuite, nous modifions donc la méthode `setUp` pour sauver la base de données avant l'exécution du test. Nous créons un objet base de données temporaire qui sera celui qui sera modifié par le test.

```
TBBlogTest >> setUp
  previousRepository := VRepository current.
  VRepository 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
```

Dans la méthode `tearDown`, à la fin de l'exécution, nous réinstallons l'objet base données sauver avant l'exécution.

```
TBBlogTest >> tearDown
  VRepository setRepository: previousRepository
```

Notez que les méthodes `setUp` et `tearDown` sont exécutées avant et après l'exécution de chaque test.

4.4 Utilisation de la base

Alors même que la base est en mémoire et bien que nous pouvons accéder au blog en utilisant le singleton de la classe `TBBlog`, nous allons montrer l'API offerte par Voyage. C'est la même API que nous pourrions utiliser pour accéder à une base Mongo.

Nous créons des posts ainsi :

```
TBBlog createDemoPosts.
```

Nous pouvons compter le nombre de blog sauvés. `count` fait partie de l'API directe de Voyage. Ici nous obtenons 1 ce qui est normal puisque le blog est implémentée comme un singleton.

```
TBBlog count
>1
```

De la même manière, nous pouvons sélectionner tous les objets sauvés.

```
TBBlog selectAll
```

On peut supprimer un objet racine en lui envoyant le message `remove`.

Vous pouvez voir l'API de Voyage en parcourant

- la classe `Class`, et
- la classe `VORepository` qui est la racine d'héritage des bases de données en mémoire ou extérieure.

Ces requêtes sont plus pertinentes quand on a plus d'objets mais nous ferions exactement les mêmes.

4.5 Si nous devons sauvegarder les posts [Discussion]

Cette section n'est pas à implémenter. Elle est juste donnée à titre de discussion (Plus d'explications sont données dans le chapitre sur Voyage dans le livre *Enterprise Pharo: a Web Perspective* disponible à <http://books.pharo.org>). Nous voulons illustrer que déclarer une classe comme une racine Voyage a une influence sur comment une instance de cette classe est sauvée et rechargée.

En particulier, déclarer un post comme une racine a comme effet que les objets posts seront des documents à part entière et ne seront plus une sous-parties d'un blog.

Lorsqu'un post n'est pas une racine, vous n'avez pas la certitude d'unicité de celui-ci lors du chargement depuis la base. En effet, lors du chargement (et ce qui peut être contraire à la situation du graphe d'objet avant la sauvegarde) un post n'est alors pas partagé entre deux instances de blogs. Si avant la sauvegarde en base un post était partagé entre deux blogs, après le chargement depuis la base, ce post sera dupliqué car recréé à partir de la définition du blog (et le blog contient alors complètement le post).

Nous pourrions définir qu'un post soit un élément qui peut être sauvegardé de manière autonome. Cela permettrait de sauver des posts de manière indépendante d'un blog.

Cependant tous les objets n'ont pas vocation être définis comme des racines. Si nous représentions les commentaires d'un post, nous ne les déclarerions pas comme racine car sauver ou manipuler un commentaire en dehors du contexte de son post ne fait pas beaucoup de sens.

Post comme racine = Unicité

Si vous désirez qu'un post soit partagé et unique entre plusieurs instances de blog, alors les objets `TBPost` doivent être déclarés comme une racine dans la base. Lorsque c'est le cas, les posts sont sauvés comme des entités autonomes et les instances de `TBLog` feront référence à ces entités au lieu que leurs définitions soient incluses dans celle des blogs. Cela a pour effet qu'un post donné devient unique et partageable via une référence depuis le blog.

Pour cela on nous définirions les méthodes suivantes:

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

Lors de l'ajout d'un post dans un blog, il est maintenant important de sauver le blog et le nouveau post.

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

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

Ici dans la méthode `removeAllPosts` nous enlevons chaque posts puis nous remettons à jour la collection.

4.6 Deployer avec une base Mongo [Optionnel]

Nous allons maintenant montrer comment utiliser une base Mongo externe à Pharo. Dans le cadre de ce tutoriel, vous pouvez ne pas le faire et passer à la suite.

En utilisant Voyage nous pouvons rapidement sauver nos posts dans une base de données Mongo. Cette section explique rapidement la mise en oeuvre et les quelques modifications que nous devons apporter à notre projet Pharo pour y parvenir.

Installation de Mongo

Quel que soit votre système d'exploitation (Linux, Mac OSX ou Windows), vous pouvez installer un serveur Mongo localement sur votre machine. Cela est pratique pour tester votre application sans avoir besoin d'une connexion Internet. Une solution consiste à installer directement un serveur Mongo sur votre système (cf. <https://www.mongodb.com>). Toutefois, nous vous conseillons plutôt d'installer Docker (<https://www.docker.com>) sur votre machine et à lancer un conteneur qui exécute un serveur Mongo grâce à la ligne de commande suivante:

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

Note Le serveur Mongo ne doit pas utiliser d'authentification (ce n'est pas le cas avec une installation locale par défaut) car la nouvelle méthode

de chiffrement SCRAM utilisée par MongoDB 3.0 n'est actuellement pas supportée par Voyage.

Quelques commandes utiles pour la suite :

```
# pour stopper votre conteneur
docker stop mongo

# pour re-démarrer votre conteneur
docker start mongo

# pour détruire votre conteneur. Ce dernier doit être stoppé avant.
docker rm mongo
```

Connexion à un serveur local

Nous définissons les méthodes `initializeLocalhostMongoDB` pour établir la connexion vers la base de données.

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

Il faut aussi s'assurer de la ré-initialisation de la connexion à la base lors du reset de la classe.

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

Vous pouvez maintenant re-créez vos posts de démo, et ils seront automatiquement sauvegardés dans votre base Mongo:

```
TBBlog reset.
TBBlog createDemoPosts
```

En cas de problème

Notez que si vous avez besoin de réinitialiser la base extérieure complètement, vous pouvez utiliser la méthode `dropDatabase`.

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

Si vous ne pouvez pas le faire depuis Pharo, vous pouvez le faire lorsque `mongod` est en cours d'exécution avec l'expression suivante :

```
[mongo tinyblog --eval "db.dropDatabase()"]
```

ou dans le conteneur docker :

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

Attention : Changements de TBBlog

Si vous utilisez une base locale plutôt qu'une base en mémoire, à chaque fois que vous déclarez une nouvelle racine d'objets ou modifiez la définition d'une classe racine (ajout, retrait, modification d'attribut) il est capital de ré-initialiser le cache maintenu par Voyage. La ré-initialisation se fait comme suit:

```
[ VORepository current reset ]
```

4.7 Conclusion

Voyage propose une API sympathique pour gérer de manière transparente la sauvegarde d'objets soit en mémoire soit dans une base de données document. Votre application peut maintenant être sauvée dans la base et vous êtes donc prêt pour construire son interface web.

Commencer avec Seaside

Dans ce chapitre nous vérifions que Seaside fonctionne et nous définissons notre premier composant. Dans les chapitres suivants, nous commençons par définir une interface telle que les utilisateurs la verrons. Dans un prochain chapitre nous développerons la logique d'identification, et finalement une interface d'administration que le possesseur du blog utilisera. Nous allons définir des composants Seaside <http://www.seaside.st> dont l'ouvrage de référence est disponible en ligne à <http://book.seaside.st>. Les premiers chapitres de <http://book.seaside.st> peuvent vous aider et compléter efficacement ce tutoriel.

Le travail présenté dans la suite est indépendant de celui sur Voyage et sur la base de données MongoDB. Vous pouvez charger le code des chapitres précédents en suivant les instructions indiquées dans le dernier chapitre (Chapitre 11).

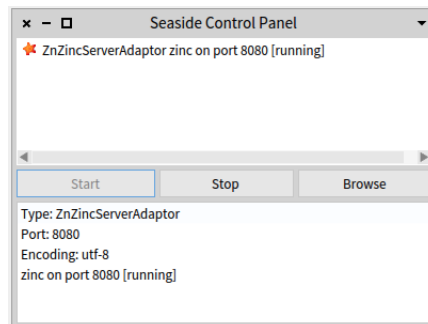


Figure 5-1 Lancer le serveur.

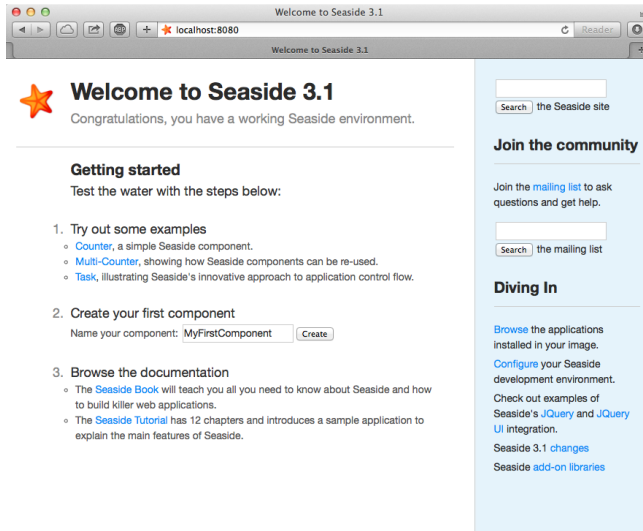


Figure 5-2 Vérification que Seaside fonctionne.

5.1 Démarrer Seaside

Seaside est déjà chargé dans l'image Pharo, sinon relisez le premier chapitre (Chapitre 11). Il existe deux façons pour démarrer Seaside. La première consiste à exécuter le code suivant :

```
[ ZnZincServerAdaptor startOn: 8080.
```

La deuxième façon est graphique via l'outil Seaside Control Panel (World Menu>Tools>Seaside Control Panel). Dans le menu contextuel de cet outil (clic droit), cliquez sur "add adaptor..." pour ajouter un serveur ZnZinc-ServerAdaptor, puis définissez le port (e.g. 8080) sur lequel le serveur doit fonctionner (comme illustré dans la figure 5-1). En ouvrant un navigateur web à l'URL <http://localhost:8080>, vous devez voir s'afficher la page d'accueil de Seaside comme sur la figure 5-2.

5.2 Bootstrap for Seaside

La bibliothèque Bootstrap est totalement accessible depuis Seaside comme nous allons le montrer. Pour parcourir les nombreux exemples, cliquer sur le lien **bootstrap** dans la liste des applications servies par Seaside ou pointer votre navigateur sur le lien <http://localhost:8080/bootstrap>. Vous devez obtenir l'écran de la figure 5-3.

Cliquer sur le lien **Examples** au bas de la page et vous pouvez ainsi voir les éléments graphiques ainsi que le code pour les obtenir comme montré par la

5.3 Point d'entrée de l'application

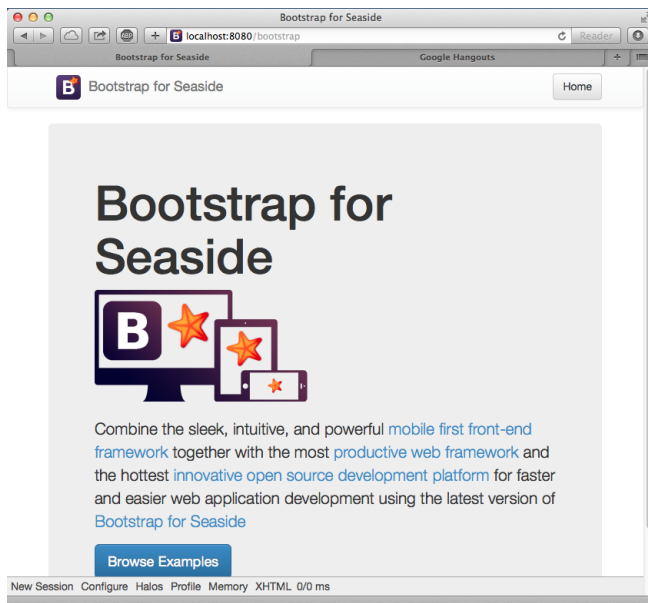


Figure 5-3 Accès à la bibliothèque Bootstrap.

figure 5-4.

Bootstrap

Le dépôt de code et la documentation de la bibliothèque Bootstrap pour Pharo est disponible ici : <http://smalltalkhub.com/#!/~TorstenBergmann/Bootstrap>. Une démo en ligne est disponible à l'adresse : <http://pharo.pharocloud.com/bootstrap>. Cette bibliothèque a déjà été chargée dans l'image PharoWeb utilisée dans ce tutoriel.

5.3 Point d'entrée de l'application

Créez la classe `TBApplicationRootComponent` qui est le point d'entrée de l'application. Elle sert à l'enregistrement de l'application au sein du serveur d'application Seaside.

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

Nous déclarons l'application au serveur Seaside, en définissant côté classe, dans le protocole 'initialization' la méthode `initialize` suivante.

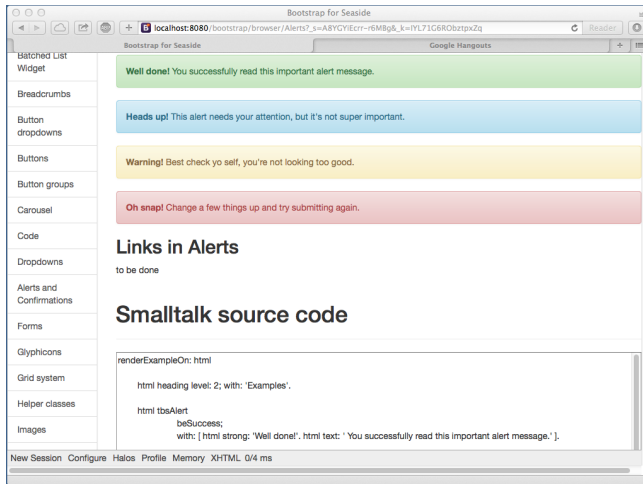


Figure 5-4 Comprendre un élément et son code en Bootstrap.

On en profite pour intégrer les dépendances du framework Bootstrap (les fichiers css et js seront stockés dans l'application).

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

Exécuter `TBApplicationRootComponent initialize` pour forcer l'exécution de la méthode `initialize`. En effet les méthodes `initialize` de classe ne sont automatiquement exécutées que l'on du chargement de la classe. Ici nous venons juste de la définir et donc il est nécessaire de l'exécuter pour en voir les bénéfices.

Ajoutons également la méthode `canBeRoot` afin de préciser que la classe `TBApplicationRootComponent` n'est pas qu'un simple composant Seaside mais qu'elle représente notre application Web. Elle sera donc instanciée dès qu'un utilisateur se connecte sur l'application.

```
TBApplicationRootComponent class >> canBeRoot
    ^ true
```

Une connexion sur le serveur Seaside ("Browse the applications installed in your image") permet de vérifier que l'application TinyBlog est bien enregistrée comme le montre la figure 5-5.

5.4 Premier rendu simple

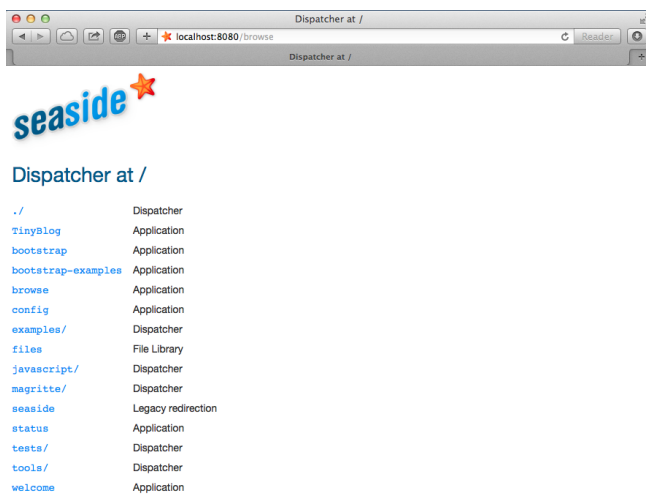


Figure 5-5 TinyBlog est bien enregistrée.

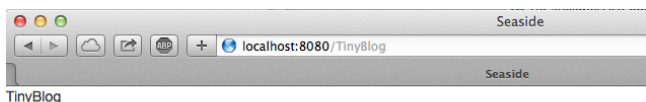


Figure 5-6 Une page quasi vide mais servie par Seaside.

5.4 Premier rendu simple

Ajoutons maintenant une méthode d'instance `renderContentOn:` dans le protocole `rendering` afin de vérifier que notre application répond bien.

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

En se connectant avec un navigateur sur `http://localhost:8080/TinyBlog`, la page affichée doit être similaire à celle sur la figure 5-6.

Ajoutons maintenant des informations dans l'entête de la page HTML afin que TinyBlog ait un titre et soit une application HTML5.

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

Le message `title:` permet de configurer le titre du document affiché dans la fenêtre du navigateur. Le composant `TBApplicationRootComponent` est le composant principal de l'application, il ne fait que du rendu graphique limité. Dans le futur, il contiendra des composants et les affichera : Comme par exemple, les composants principaux de l'application permettant l'affichage des posts pour les lecteurs du blog mais également des composants pour administrer le blog et ses posts.

5.5 Architecture

Nous sommes maintenant prêts à définir les composants visuels de notre application Web. Afin de vous donner une vision d'ensemble nous montrons comment chaque composant est responsable d'une partie de l'application (Figure 5-7) et comment ces composants s'intègre au sein de l'architecture (simple) du système (Figure 5-8).

Premier aperçu des composants

La figure 5-7 montre les différents composants que nous allons développer et où ils se situent.

Éléments

Pour vous permettre de suivre le développement incrémental de l'application, nous décrivons son architecture dans la figure 5-8.

- `ApplicationRootComponent` est le point d'entrée pour le serveur d'application. Ce composant contient des composants héritant tous de la classe abstraite `ScreenComponent` que nous allons définir dans le prochain chapitre.
- `ScreenComponent` est la racine d'héritage des composants utilisateurs ou admin. Il est composé d'un header.
- `PostsListComponent` est le composant principal affichant les bulletins (posts). Il est composé d'instances de `PostComponent` et gère aussi les catégories.
- `AdminComponent` est le composant principal de la partie administration. Il contient un rapport (instance de `PostsReport`) sur les bulletins (posts) construits en utilisant Magritte.

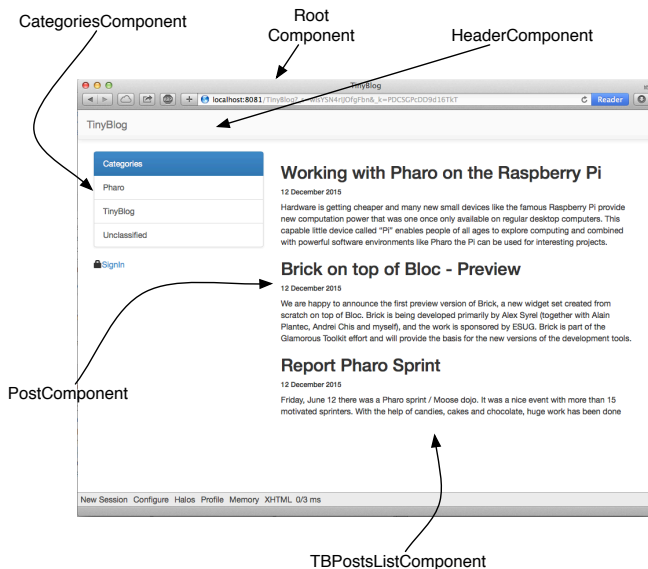


Figure 5-7 Les composants composant l'application TinyBlog (en mode non admin).

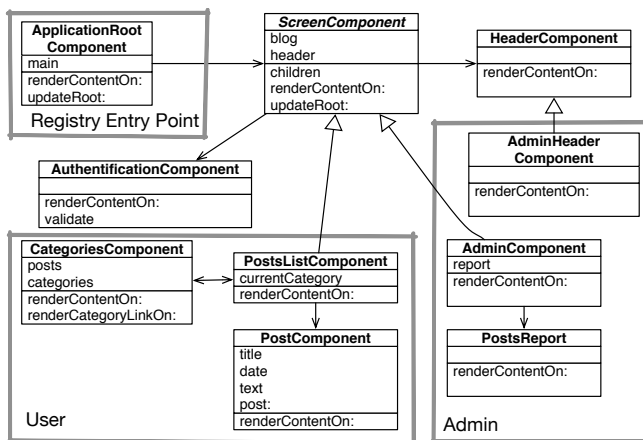


Figure 5-8 Architecture de TinyBlog.

5.6 Conclusion

Nous sommes prêts à développer les composants décrits. Comme le processus est un peu linéaire n'hésitez pas à revenir sur cette architecture afin de comprendre le composant que vous nous proposons de développer.

Des composants web pour TinyBlog

Dans ce chapitre, commençons par définir une interface publique permettant d'afficher les bulletins (posts) du blog. Nous raffinons cela dans le chapitre suivant.

Si vous avez le sentiment d'être un peu perdu, la figure 6-1 vous montre les composants sur lesquels nous allons travailler dans ce chapitre.

Le travail présenté dans la suite est indépendant de celui sur Voyage et sur la base de données MongoDB. Les instructions pour charger le code des chapitres

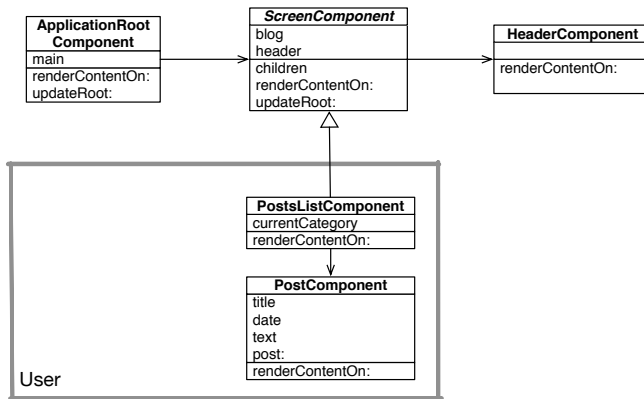


Figure 6-1 L'architecture des composants utilisateurs (par opposition à administration).

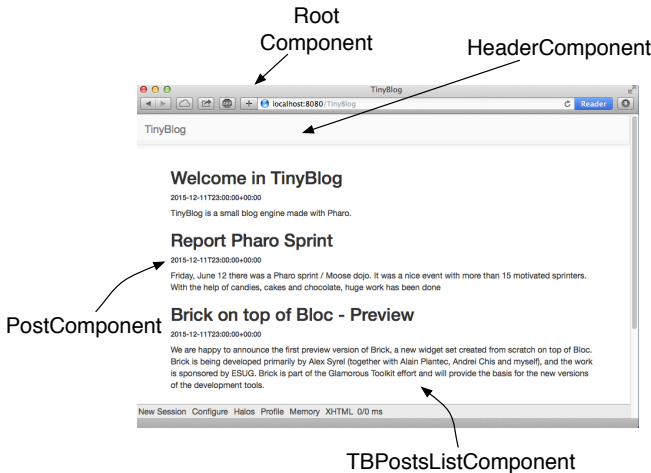


Figure 6-2 Les composants visuels de l'application TinyBlog.

précédents sont disponibles dans le chapitre 11.

6.1 Composants visuels

Nous sommes maintenant prêts à définir les composants visuels de notre application Web. La figure 6-2 montre les différents composants que nous allons développer dans ce chapitre et où ils se situent.

Le composant TBScreenComponent

Le composant `TBApplicationRootComponent` contiendra des composants sous-classes de la classe abstraite `TBScreenComponent`. Cette classe nous permet de factoriser les comportements que nous souhaitons partager entre tous nos composants.

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

Les différents composants d'interface de TinyBlog ont besoin d'accéder aux règles métier de l'application. Dans le protocole 'accessing', créons une méthode `blog` qui retourne une instance de `TBBlog` (ici notre singleton). Notez que cette méthode pourrait renvoyer l'instance de blog avec laquelle elle a été configurée au préalable.

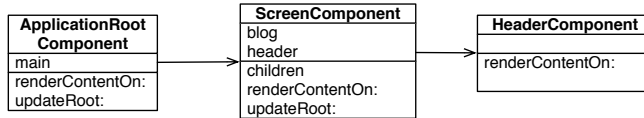


Figure 6-3 Le composant `ApplicationRootComponent` utilise de manière temporaire le composant `ScreenComponent` qui a un `HeaderComponent`.

```

[ 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

```

Par la suite, si l'on souhaite étendre `TinyBlog` pour qu'un utilisateur puisse avoir plusieurs blogs, il suffira de modifier cette méthode pour utiliser des informations stockées dans la session active (Voir `TBSession` dans le chapitre suivant).

Définissez la méthode `renderContentOn:` de ce composant comme suit temporairement. Notez que pour l'instant, nous n'affichons pas ce composant donc rafraîchir votre browser ne vous montre rien de nouveau et c'est normal.

```

[ TBScreenComponent >> renderContentOn: html
  html text: 'Hello from TBScreenComponent'

```

6.2 Utilisation du composant Screen

Bien que le composant `TBScreenComponent` n'ait pas vocation à être utilisé directement, nous allons l'utiliser de manière temporaire pendant que nous développons les autres composants.

Nous ajoutons la variable d'instance `main` dans la classe `TBApplicationRootComponent`.

```

[ WAComponent subclass: #TBApplicationRootComponent
  instanceVariableNames: 'main'
  classVariableNames: ''
  package: 'TinyBlog-Components'

```

Nous initialisons cette variable d'instance dans la méthode `initialize` suivante et redéfinissons la méthode `children`. Nous obtenons la situation décrite par la figure 6-3.

```

[ TBApplicationRootComponent >> initialize
  super initialize.
  main := TBScreenComponent new

```

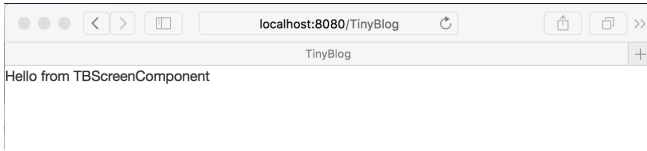


Figure 6-4 Premier visuel du composant TBScreenComponent.

```
[ TBAplicationRootComponent >> renderContentOn: html
  html render: main
```

Nous déclarons aussi la relation de contenu en retournant le composant référencé par la variable `main` parmi les enfants de `TBAplicationRootComponent`.

```
[ TBAplicationRootComponent >> children
  ^ { main }
```

Si vous rafraîchissez votre browser, vous allez voir l’affichage produit par le sous-composant `TBScreenComponent` qui affiche pour l’instant le texte: `Hello from TBScreenComponent` (voir la figure 6-4).

6.3 Pattern de définition de composants

Nous allons souvent utiliser la même façon de procéder:

- nous définissons d’abord la classe et le comportement d’un nouveau composant;
- puis, nous allons y faire référence depuis la classe qui utilisera ce composant pour satisfaire les contraintes de Seaside;
- en particulier, nous exprimons la relation entre un composant et un sous-composant en redéfinissant la méthode `children`.

6.4 Ajouter quelques bulletins au blog

Vérifiez que votre blog a quelques bulletins :

```
[ TBlog current allBlogPosts size
```

Si il n’en contient aucun, recréez-en :

```
[ TBlog createDemoPosts
```


6.5 Définition du composant TBHeaderComponent

Définissons une en-tête commune à toutes les pages de TinyBlog dans un composant nommé TBHeaderComponent. Ce composant sera inséré dans la partie supérieure de chaque composant (TBPostsListComponent par exemple). Nous appliquons le schéma décrit ci-dessus: définition d'une classe, référence depuis la classe utilisatrice, et redéfinition de la méthode children.

Nous définissons d'abord sa classe, puis nous allons y faire référence depuis la classe qui l'utilise. Ce faisant, nous allons montrer comment un composant exprime sa relation à un sous-composant.

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

6.6 Utilisation du composant header

Complétons maintenant la classe TBScreenComponent afin qu'elle affiche une instance de TBHeaderComponent. Pour rappel, TBScreenComponent est la super-classe abstraite (nous l'utilisons directement pour l'instant) de tous nos composants dans l'architecture finale. Cela signifie que toutes les sous-classes de TBScreenComponent seront des composants avec une en-tête. Pour éviter d'instancier systématiquement le composant TBHeaderComponent à chaque fois qu'un composant est appelé, créons et initialisons une variable d'instance header dans TBScreenComponent.

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

Créons une méthode initialize dans le protocole 'initialization':

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

TBScreenComponent >> createHeaderComponent
  ^ TBHeaderComponent new
```

Notez que nous avons une méthode spécifique pour créer le composant en-tête. Nous pouvons ainsi redéfinir cette méthode afin de changer le composant en-tête. Cela sera utile pour la partie administration du site.

6.7 Relation composite-composant

En Seaside, les sous-composants d'un composant doivent être retournés par le composite en réponse au message `children`. Définissons que l'instance du composant `TBHeaderComponent` est un enfant de `TBScreenComponent` dans la hiérarchie des composants Seaside (et non entre classes Pharo). Dans cet exemple, nous spécialisons la méthode `children` pour qu'elle retourne une collection contenant un seul élément qui est l'instance de `TBHeaderComponent` référencée depuis la variable `header`.

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

6.8 Rendu visuel de la barre de navigation

Affichons maintenant le composant dans la méthode `renderContentOn:` (protocole 'rendering') :

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

Si vous rafraichissez votre navigateur web, rien ne se passe car le composant `TBHeaderComponent` n'a pas de rendu visuel. Pour cela, définissons la méthode `renderContentOn:` chargée d'afficher l'en-tête comme suit :

```
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' ]
```

L'entête (header) est affichée à l'aide d'une barre de navigation Bootstrap. Si vous faites un rafraichissement de l'application dans votre navigateur web vous devez voir apparaître l'en-tête comme sur la figure 6-5.

Par défaut dans une barre de navigation Bootstrap, il y a un lien sur le titre de l'application (`tbsNavbarBrand`) qui permet de revenir à la page de départ du site.

Améliorations possibles

Le nom du blog devrait être paramétrable à l'aide d'une variable d'instance dans la classe `TBBlog` et l'en-tête pourrait afficher ce titre.

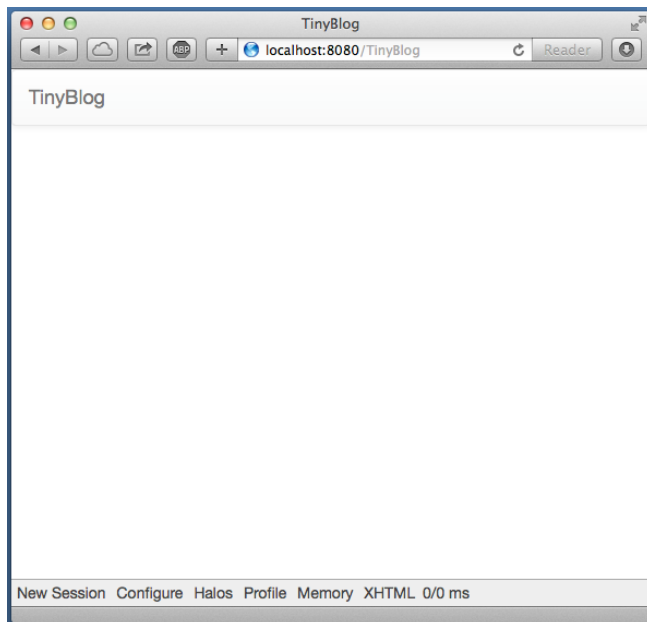


Figure 6-5 TinyBlog avec une barre de navigation.

6.9 Liste des posts

Créons un composant `TBPostsListComponent` pour afficher la liste des bulletins (posts) - ce qui reste d'ailleurs le but d'un blog. Ce composant constitue la partie publique du blog offerte aux lecteurs du blog.

Pour cela, définissons une sous-classe de `TBScreenComponent` (comme illustré dans la figure 6-6):

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

Nous pouvons maintenant modifier le composant principal de l'application (`TBApplicationRootComponent`) pour qu'il affiche ce nouveau composant. Pour cela nous modifions sa méthode `initialize` ainsi:

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

Ajoutons également une méthode `setter` (`main:`) qui nous permettra par la suite, de changer dynamiquement le sous-composant à afficher tout en gardant le composant actuel (instance de `TBPostsListComponent`) par défaut.

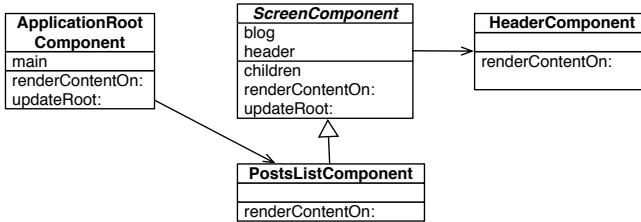


Figure 6-6 Le composant `ApplicationRootComponent` utilise le composant `PostsListComponent`.

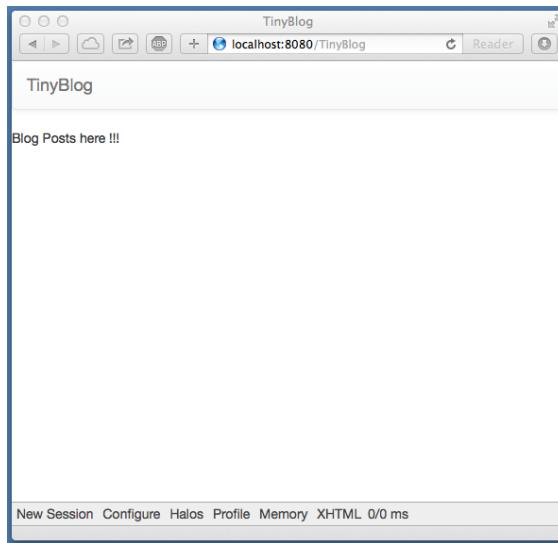


Figure 6-7 TinyBlog avec une liste de bulletins plutôt élémentaire.

```

[ TBAApplicationRootComponent >> main: aComponent
  main := aComponent

```

Ajoutons une méthode `renderContentOn:` (protocole rendering) provisoire pour tester l'avancement de notre application (voir figure 6-7). Notez que cette méthode fait un appel à la méthode `renderContentOn:` de la super-classe qui va afficher le composant en-tête.

```

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

```

Si vous rafraîchissez la page de TinyBlog dans votre navigateur, vous devriez obtenir la même chose que sur la figure 6-7.

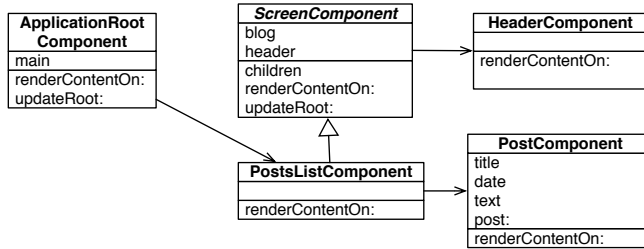


Figure 6-8 Ajout du composant Post.

6.10 Le composant Post

Nous allons maintenant définir le composant `TBPostComponent` qui affiche le contenu d'un bulletin (post).

Chaque bulletin du blog sera représenté visuellement par une instance de `TBPostComponent` qui affiche le titre, la date et le contenu d'un bulletin. Nous allons obtenir la situation décrite par la figure 6-8.

```

[ WComponent subclass: #TBPostComponent
  instanceVariableNames: 'post'
  classVariableNames: ''
  package: 'TinyBlog-Components'

```

```

[ TBPostComponent >> initialize
  super initialize.
  post := TBPost new

```

```

[ TBPostComponent >> title
  ^ post title

```

```

[ TBPostComponent >> text
  ^ post text

```

```

[ TBPostComponent >> date
  ^ post date

```

Ajoutons la méthode `renderContentOn:` qui définit l'affichage du post.

```

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

```

A propos des formulaires

Dans le chapitre sur l'interface d'administration, nous utiliserons Magritte et montrerons qu'il est rare de définir un composant de manière aussi manuelle

comme ci-dessus. En effet, Magritte permet de décrire les données manipulées et offre ensuite la possibilité de générer automatiquement des composants Seaside. Le code équivalent à celui ci-dessus en Magritte serait comme suit:

```
[ TBPPostComponent >> renderContentOn: html
  "DON'T WRITE THIS YET"
  html render: post asComponent
```

6.11 Afficher les bulletins (posts)

Il ne reste plus qu'à modifier la méthode `renderContentOn:` de la classe `TBPostsListComponent` pour afficher l'ensemble des bulletins visibles présents dans la base.

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

Rafraîchissez la page de votre navigateur et vous devez obtenir une page d'erreur.

6.12 Débugger les erreurs

Par défaut, lorsqu'une erreur se produit dans une application, Seaside retourne une page HTML contenant un message. Vous pouvez changer ce message, mais le plus pratique pendant le développement de l'application est de configurer Seaside pour qu'il ouvre un debugger dans Pharo. Pour cela, exécuter le code suivant :

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

Rafraîchissez la page de votre navigateur et vous devez obtenir un debugger côté Pharo. L'analyse de la pile d'appels montre qu'il manque la méthode suivante :

```
[ TBPPostComponent >> post: aPost
  post := aPost
```

Vous pouvez ajouter cette méthode dans le debugger avec le bouton **Create**. Quand c'est fait, appuyez sur le bouton **Proceed**. La page de votre navigateur doit maintenant montrer la même chose que la figure 6-9.

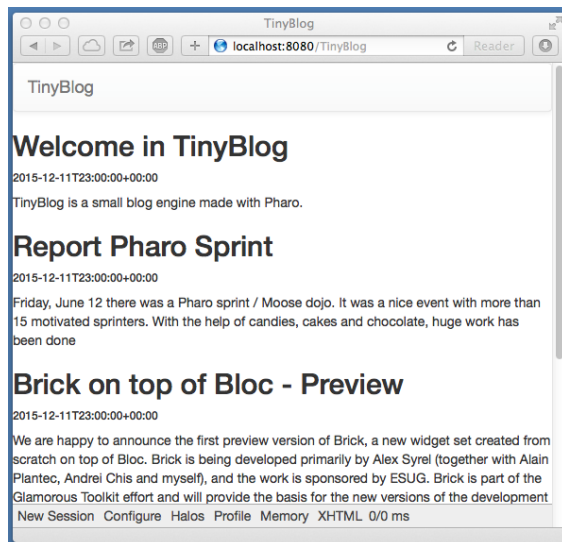


Figure 6-9 TinyBlog avec une liste de posts.

6.13 Affichage de la liste des posts avec Bootstrap

Nous allons utiliser Bootstrap pour rendre la liste un peu plus jolie à l'aide d'un container en utilisant le message `tbsContainer:` comme suit :

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

Rafraichissez la page et vous devez obtenir la figure 6-2.

6.14 Cas d'instanciation de composants dans `renderContentOn:`

Nous avons dit que la méthode `children` d'un composant devait retourner ses sous-composants. En effet, avant d'exécuter la méthode `renderContentOn:` d'un composite, Seaside a besoin de retrouver tous les sous-composants de ce composite et notamment leurs états.

Toutefois, si des sous-composants sont instanciés systématiquement dans la méthode `renderContentOn:` du composite, comme c'est le cas dans la méthode `renderContentOn:` de la classe `TBPostsListComponent` ci-dessus, il n'est pas nécessaire qu'ils soient stockés et retournés par la méthode `children`.

dren du composite. Bien évidemment, instancier systématiquement des sous-composants dans la méthode `renderContentOn` : n'est pas forcément une bonne pratique car cela allonge le délai de rendu d'une page Web.

Si nous voulions stocker les sous-composants permettant d'afficher les bulletins, nous aurions ajouté et initialisé une variable d'instance `postComponents`.

```
TBPostsListComponent >> initialize
  super initialize.
  postComponents := OrderedCollection new
```

Nous aurions ajouté la méthode `postComponents` calculant les composants pour les bulletins.

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

Et nous aurions finalement modifié la méthode `children` et `renderContentOn`:

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

TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html tbsContainer: [
    self postComponents do: [ :p |
      html render: p ] ]
```

Nous ne le faisons pas car cela complique le code et n'apporte pas grand chose puisque les sous-composants sont tout de même instanciés à chaque rendu du composant `TBPostsListComponent`.

6.15 Conclusion

Nous avons développé le rendu d'une liste de bulletins et dans le chapitre suivant nous allons ajouter la gestion des catégories.

Avec Seaside, le programmeur n'a pas à se soucier de gérer les requêtes web, ni l'état de l'application. Il définit des composants qui sont créés et sont proches des composants pour applications de bureau.

Un composant Seaside est responsable d'assurer son rendu en spécialisant la méthode `renderContentOn` :. De plus un composant doit retourner ses sous-composants en spécialisant la méthode `children`.

Gestion des catégories

Dans ce chapitre, nous allons ajouter la gestion des catégories des bulletins. Si vous avez le sentiment d'être un peu perdu, la figure 7-1 vous montre les composants sur lesquels nous allons travailler dans ce chapitre.

Les instructions pour charger le code des chapitres précédents sont disponibles dans le chapitre 11.

7.1 Affichage des bulletins par catégorie

Les bulletins sont classés par catégorie. Par défaut, si aucune catégorie n'a été précisée, ils sont rangés dans une catégorie spéciale dénommée "Unclas-

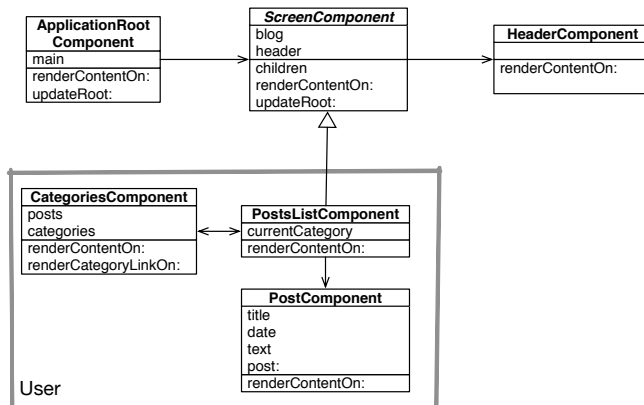


Figure 7-1 L'architecture des composants de la partie publique avec catégories.

sified”. Nous allons créer un composant nommé `TBCategoriesComponent` pour gérer une liste de catégories.

Pour afficher les catégories

Nous avons besoin d'un composant qui affiche la liste des catégories présentes dans le blog et permet d'en sélectionner une. Ce composant devra donc avoir la possibilité de communiquer avec le composant `TBPostsListComponent` afin de lui communiquer la catégorie courante. La situation est décrite par la figure 7-1.

Rappelez-vous qu'une catégorie est simplement exprimée comme une chaîne de caractères dans le modèle défini dans le Chapitre 2 et comme l'illustre le test suivant.

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

Définition du composant

Nous définissons un nouveau composant nommé `TBCategoriesComponent`. Ce composant va garder une collection triée par ordre alphabétique de chaînes de caractères pour chacune des catégories ainsi qu'un pointeur sur le composant `postsList` associé.

```
[ WAComponent subclass: #TBCategoriesComponent
  instanceVariableNames: 'categories postsList'
  classVariableNames: ''
  package: 'TinyBlog-Components'
```

Nous définissons les accesseurs associés.

```
[ TBCategoriesComponent >> categories
  ^ categories

[ TBCategoriesComponent >> categories: aCollection
  categories := aCollection asSortedCollection

[ TBCategoriesComponent >> postsList: aComponent
  postsList := aComponent

[ TBCategoriesComponent >> postsList
  ^ postsList
```

Nous définissons aussi une méthode de création au niveau classe.

```
[ TBCategoriesComponent class >> categories: categories postsList:
  aTBScreen
  ^ self new categories: categories; postsList: aTBScreen
```

Liaison depuis la liste de bulletins

Nous avons donc besoin d'ajouter une variable d'instance pour stocker la catégorie courante dans la classe `TBPostsListComponent`.

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

Nous définissons les accesseurs associés.

```
TBPostsListComponent >> currentCategory
  ^ currentCategory

TBPostsListComponent >> currentCategory: anObject
  currentCategory := anObject
```

La méthode `selectCategory:`

La méthode `selectCategory:` (protocole 'actions') communique au composant `TBPostsListComponent` la nouvelle catégorie courante.

```
TBCategoriesComponent >> selectCategory: aCategory
  postsList currentCategory: aCategory
```

7.2 Rendu des catégories

Nous ajoutons la méthode `renderCategoryLinkOn:with:` (protocole 'rendering') pour afficher les catégories sur la page. En particulier, pour chaque catégorie nous définissons le fait que cliquer sur la catégorie la sélectionne comme la catégorie courante. Nous utilisons un callback (message `callback:`). L'argument de ce message est un block qui peut contenir n'importe quelle expression Pharo. Cela illustre la puissance de Seaside.

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

La méthode de rendu `renderContentOn:` du composant `TBCategoriesComponent` est simple : on itère sur toutes les catégories et on les affiche en utilisant Bootstrap.

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

Nous avons presque fini mais il faut encore afficher la liste des catégories et mettre à jour la liste des bulletins en fonction de la catégorie courante.

7.3 Mise à jour de la liste des bulletins

Nous devons mettre à jour les bulletins. Pour cela, modifions la méthode de rendu du composant `TBPostsListComponent`.

La méthode `readSelectedPosts` récupère les bulletins à afficher depuis la base et les filtre en fonction de la catégorie courante. Si la catégorie courante est nil, cela signifie que l'utilisateur n'a pas encore sélectionné de catégorie et l'ensemble des bulletins visibles de la base est affiché. Si elle contient une valeur autre que nil, l'utilisateur a sélectionné une catégorie et l'application affiche alors la liste des bulletins attachés à cette catégorie.

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

Nous pouvons maintenant modifier la méthode chargée du rendu de la liste des posts :

```
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) ] ]
```

Une instance du composant `TBCategoriesComponent` est ajoutée sur la page et permet de sélectionner la catégorie courante (voir la figure 7-2). De même qu'expliqué précédemment, une nouvelle instance de `TBCategoriesComponent` est créée à chaque rendu du composant `TBPostsListComponent`, donc il n'est pas nécessaire de l'ajouter dans la liste des sous-composants retourné par `children`.

Améliorations possibles

Mettre en dur le nom des classes et la logique de création des catégories et des bulletins n'est pas optimale. Proposer quelques méthodes pour résoudre cela.

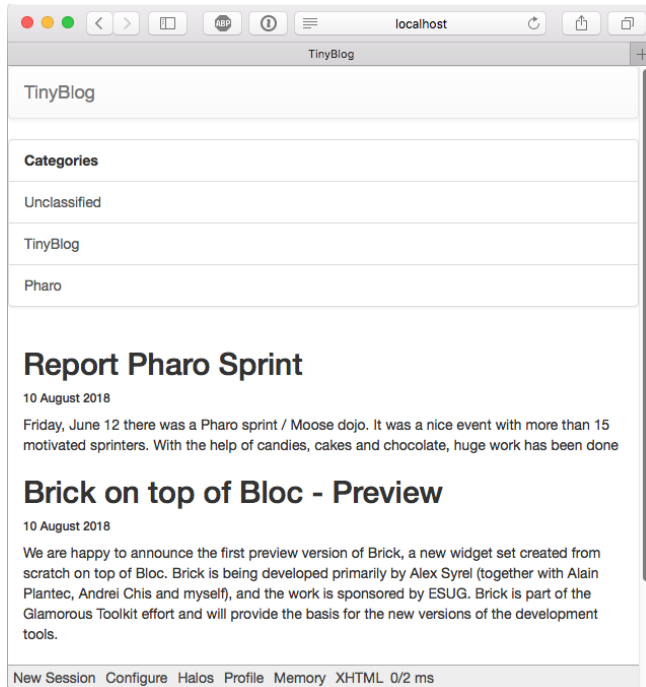


Figure 7-2 Catégories afin de sélectionner les posts.

7.4 Look et agencement

Nous allons maintenant agencer le composant `TBPostsListComponent` en utilisant un 'responsive design' pour la liste des bulletins (voir la figure 7-3). Cela veut dire que le style CSS va adapter les composants à l'espace disponible.

Les composants sont placés dans un container Bootstrap puis agencés sur une ligne avec deux colonnes. La dimension des colonnes est déterminée en fonction de la résolution (viewport) du terminal utilisé. Les 12 colonnes de Bootstrap sont réparties entre la liste des catégories et la liste des posts. Dans le cas d'une résolution faible, la liste des catégories est placée au dessus de la liste des posts (chaque élément occupant 100% de la largeur du container).

```
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html tbsContainer: [
    html tbsRow showGrid;
    with: [
      html tbsColumn
        extraSmallSize: 12;
        smallSize: 2;
    ]
  ]
```

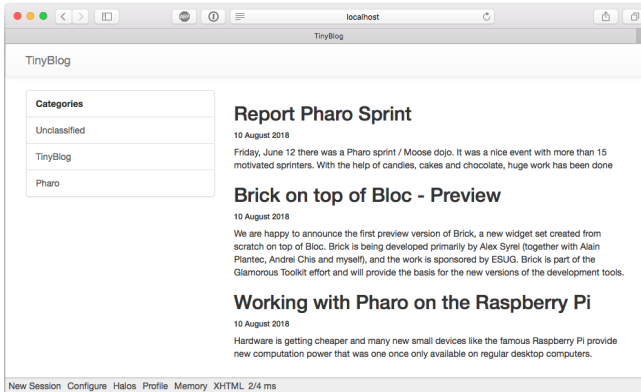


Figure 7-3 Liste des catégories avec un meilleur agencement.

```

        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) ] ] ] ]

```

Vous devez obtenir une application proche de celle représentée par la figure 7-3.

Lorsqu'on sélectionne une catégorie, la liste des posts est bien mise à jour. Toutefois, l'entrée courante dans la liste des catégories n'est pas sélectionnée. Pour cela, on modifie la méthode suivante :

```

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

```

Bien que le code fonctionne, on ne doit pas laisser la méthode `renderContentOn:` de la classe `TBPostsListComponent` dans un tel état. Elle est bien trop longue et difficilement réutilisable. Proposer une solution.

7.5 Modulariser son code avec des petites méthodes

Voici notre solution au problème précédent. Pour permettre une meilleure lecture et réutilisation future, nous commençons par définir les méthodes de création des composants

```

TbPostsListComponent >> categoriesComponent
^ TBCategoriesComponent
  categories: self blog allCategories
  postsList: self

TbPostsListComponent >> postComponentFor: aPost
^ TBPostComponent new post: aPost

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) ]
        ] ] ] ]

```

Autre passe de découpage

Continuons à découper cette méthode en plusieurs petites méthodes. Pour cela, créons des méthodes pour les traitements élémentaires.

```

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

```

Puis nous utilisons ces traitements pour simplifier la méthode `renderContentOn`:

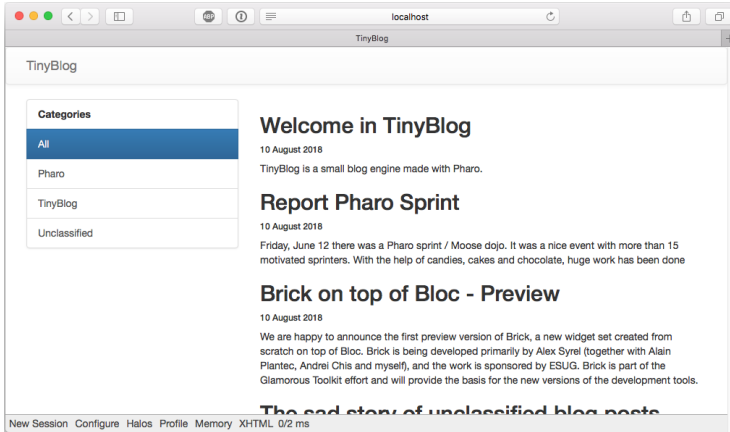


Figure 7-4 TinyBlog UI version finale.

```
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html
    tbsContainer: [
      html tbsRow
        showGrid;
        with: [ self renderCategoryColumnOn: html.
                self renderPostColumnOn: html ] ]

TBPostsListComponent >> renderCategoryColumnOn: html
  html tbsColumn
    extraSmallSize: 12;
    smallSize: 2;
    mediumSize: 4;
    with: [ self basicRenderCategoriesOn: html ]

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

L'application finale est affichée dans la figure 7-4.

7.6 Conclusion

Nous avons défini une interface pour notre blog en utilisant un ensemble de composants définissant chacun leur propre état et leurs responsabilités. Maintenant il faut remarquer que de très nombreuses applications se con-

struisent de la même manière. Donc vous avez les bases pour définir de nombreuses applications web.

Dans le prochain chapitre, nous allons voir comment gérer l'identification permettant d'accéder à la partie administration des bulletins.

Améliorations possibles

A titre d'exercice, vous pouvez :

- trier les catégories par ordre alphabétique ou
- ajouter un lien nommé 'All' dans la liste des catégories permettant d'afficher toutes les posts visible quelque soit leur catégorie.

Authentification et Session

Le scénario assez classique que nous allons développer dans ce chapitre est le suivant : l'utilisateur doit s'authentifier pour accéder à la partie administration de TinyBlog. Il le fait à l'aide d'un compte et d'un mot de passe.

La figure 8-1 montre un aperçu de l'architecture visée dans ce chapitre.

Nous commençons par mettre en place une première version permettant de naviguer entre la partie publique TinyBlog rendue par le composant gérant la liste des bulletins (TBPostsListComponent) et une première version de la partie d'administration du site comme sur la figure 8-2. Cela va nous permettre d'illustrer l'invocation de composant.

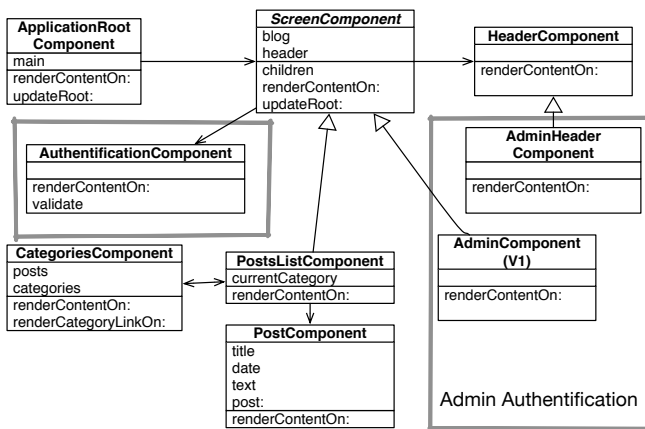


Figure 8-1 Gérant l'authentification pour accéder à l'administration.

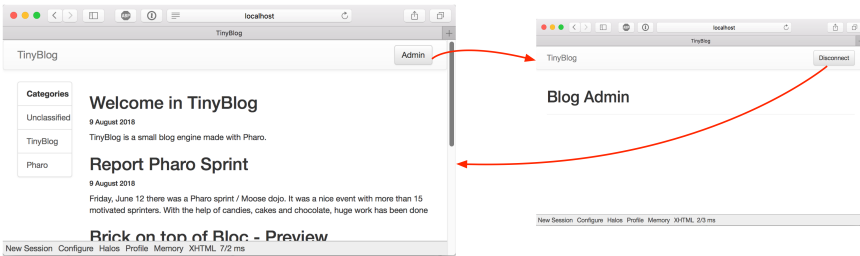


Figure 8-2 Lien simple vers la partie administration.

Nous intégrerons ensuite un composant d'identification sous la forme d'une boîte modale. Cela va nous permettre d'illustrer comment la saisie de champs utilise de manière élégante les variables d'instances d'un composant.

Enfin, nous montrerons aussi comment stocker l'utilisateur connecté dans la session courante.

8.1 Composant d'administration simple (v1)

Définissons un composant d'administration très simple. Ce composant hérite de la classe `TBScreenComponent` comme mentionné dans un chapitre précédent et illustré dans la figure 8-1.

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

Nous définissons une première version de la méthode de rendu afin de pouvoir tester.

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

8.2 Ajout d'un bouton 'admin'

Ajoutons maintenant un bouton dans l'en-tête du site (composant `TBHeaderComponent`) afin d'accéder à la partie administration du site comme sur la figure 8-2. Pour cela, modifions les composants existants: `TBHeaderComponent` (en-tête) et `TBPostsListComponent` (partie publique).

Commençons par ajouter le bouton 'admin' dans l'en-tête :

8.3 Revisons la barre de navigation

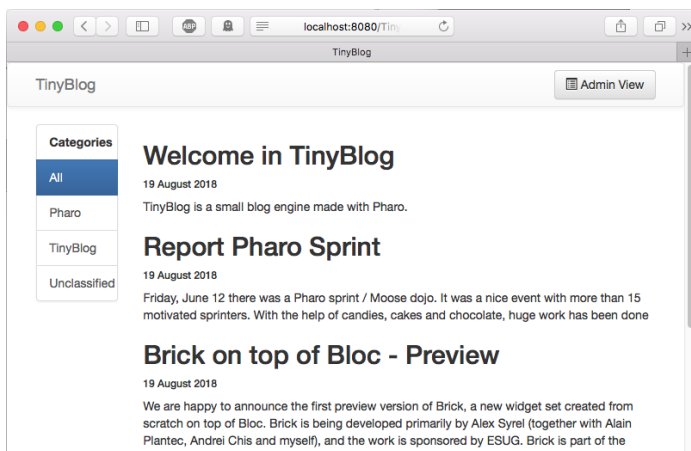


Figure 8-3 Barre de navigation avec un button admin.

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

Si vous rafraichissez votre navigateur, le bouton admin est bien présent mais il n'a aucun effet pour l'instant (voir la figure 8-3). Il faut définir un callback: sur ce bouton (un bloc) qui remplace le composant courant (TBPostsListComponent) par le composant d'administration (TBAAdminComponent).

8.3 Revisons la barre de navigation

Commençons par réviser la définition de TBHeaderComponent en lui ajoutant une variable d'instance component pour stocker et accéder au composant courant (qui sera soit la liste de bulletin, soit le composant d'administration).

Ceci va nous permettre de pouvoir accéder au composant depuis la barre de navigation :

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

TBHeaderComponent >> component: anObject
  component := anObject

TBHeaderComponent >> component
  ^ component
```

Nous ajoutons une méthode de classe.

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

8.4 Activation du button d'admin

Modifions l'instanciation du composant en-tête définie dans la méthode du composant `TBScreenComponent` afin de passer le composant qui sera sous la barre de navigation à celle-ci :

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

Notez que la méthode `createHeaderComponent` est bien définie dans la superclasse `TBScreenComponent` car elle est applicable pour toutes ses sousclasses.

Nous pouvons maintenant ajouter le callback (message `callback:`) sur le bouton :

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

Il ne reste plus qu'à définir la méthode `goToAdministrationView` sur le composant `TBPostsListComponent` dans le protocole 'actions' :

```
TBPostsListComponent >> goToAdministrationView
  self call: TBAdminComponent new
```

8.5 Ajout d'un bouton 'disconnect'

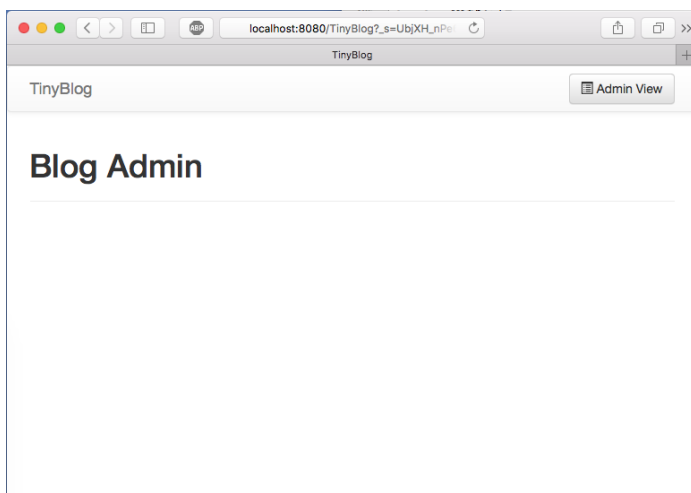


Figure 8-4 Affichage du composant admin en cours de définition.

Avant de cliquer sur le bouton 'Admin' dans votre navigateur, vous devez cliquer sur 'New Session' afin de recréer le composant TBHeaderComponent. Vous devez obtenir la situation présentée dans la figure 8-4. Le bouton 'Admin' permet maintenant de voir la partie administration v1 s'afficher. Attention à ne cliquer qu'une seule fois car ce bouton 'Admin' est toujours présent dans la partie administration bien qu'il ne soit pas fonctionnel. Nous allons le remplacer par un bouton 'Disconnect'.

8.5 Ajout d'un bouton 'disconnect'

Lorsqu'on affiche la partie administration, nous allons remplacer le composant en-tête par un autre. Cette nouvelle en-tête affichera un bouton 'Disconnect'.

Définissons un nouveau composant en-tête:

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

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

Indiquons au composant TBAdminComponent d'utiliser cette en-tête :

```
TBAdminComponent >> createHeaderComponent
  ^ TBAdminHeaderComponent from: self
```

Maintenant nous pouvons spécialiser notre nouvelle barre de navigation dédiée à l'administration pour afficher un bouton de déconnexion.

```
TBAdminHeaderComponent >> renderDisconnectButtonOn: html
  html tbsNavbarButton
    tbsPullRight;
    callback: [ component goToPostListView ];
    with: [
      html text: 'Disconnect '.
      html tbsGlyphIcon iconLogout ]
TBAdminComponent >> goToPostListView
  self answer
```

Le message `answer` donne le contrôle au composant qui l'a invoqué. Ici nous retournons donc à la liste de bulletins.

Cliquez sur 'New Session' en bas à gauche de votre navigateur et ensuite sur le bouton 'Admin', vous devez maintenant voir la partie administration v1 s'afficher avec un bouton 'Disconnect' permettant de revenir à la partie publique comme sur la figure 8-2.

Notion `call:/answer`:

Si vous étudiez le code précédent, vous verrez que nous avons utilisé le mécanisme `call:/answer`: de Seaside pour mettre en place la navigation entre les composants `TBPostsListComponent` et `TBAdminComponent`. Le message `call:` remplace le composant courant par le composant passé en argument et lui donne le flot de calcul. Les message `answer`: retourne une valeur à cet appel et redonne le contrôle au composant appelant. Ce mécanisme puissant et élégant est expliqué dans la vidéo 1 de la semaine 5 du Mooc (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_5.html).

8.6 Composant fenêtre modale d'identification

Développons maintenant un composant d'identification qui lorsqu'il sera invoqué ouvrira une boîte de dialogue pour demander un login et un mot de passe. Le résultat que nous voulons obtenir est montré sur la figure 8-5.

Sachez qu'il existe des bibliothèques de composants Seaside prêt à l'emploi. Par exemple, le projet Heimdal disponible sur <http://www.github.com/DuneSt/> offre un composant d'identification ou le projet Steam <https://github.com/guillep/steam> offre d'autres composants permettant d'interroger google ou twitter.

8.6 Composant fenêtre modale d'identification

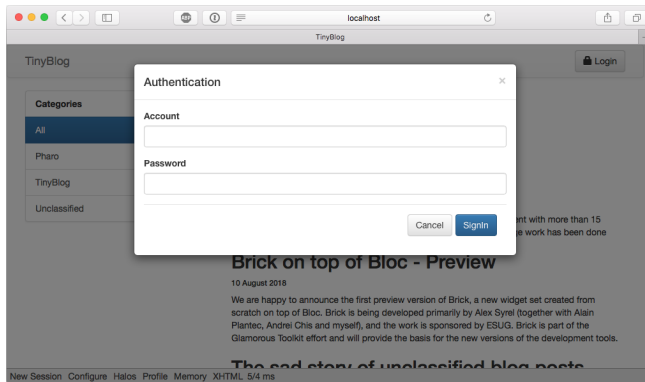


Figure 8-5 Aperçu du composant d'identification.

Définition d'un composant d'identification

Nous définissons une nouvelle sous-classe de la classe `WComponent` et des accesseurs. Ce composant contient un login, un mot de passe ainsi que le composant qui l'a invoqué pour accéder à la partie administration.

```
[ WComponent 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
```

La variable d'instance `component` sera initialisée par la méthode de classe suivante :

```
[ TBAuthenticationComponent class >> from: aComponent
  ^ self new
    component: aComponent;
    yourself
```

8.7 Rendu du composant d'identification

La méthode `renderContentOn:` suivante définit le contenu d'une boîte de dialogue modale avec l'identifiant `myAuthDialog`. Cet identifiant sera utilisé pour sélectionner le composant qui sera rendu visible en mode modal plus tard. Cette boîte de dialogue est composée d'une en-tête et d'un corps. Notez l'utilisation des messages `tbsModal`, `tbsModalBody:` et `tbsModalContent:` qui permettent une interaction modale avec ce composant.

```
TBAuthenticationComponent >> renderContentOn: html
    html tbsModal
      id: 'myAuthDialog';
      with: [
        html tbsModalDialog: [
          html tbsModalContent: [
            self renderHeaderOn: html.
            self renderBodyOn: html ] ] ]
```

L'en-tête affiche un bouton pour fermer la boîte de dialogue et un titre avec de larges fontes. Notez que vous pouvez également utiliser la touche `esc` du clavier pour fermer la fenêtre modale.

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

Le corps du composant affiche un masque de saisie pour l'identifiant, le mot de passe et finalement des boutons.

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

La méthode `renderAccountFieldOn:` montre comment la valeur d'un input field est passée puis stockée dans une variable d'instance du composant quand l'utilisateur confirme sa saisie. Le paramètre de la méthode `callback:` est un bloc qui prend lui-même un argument représentant la valeur du text input.

```

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

```

Le même procédé est utilisé pour le mot de passe.

```

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

```

Deux boutons sont ajoutés en bas de la fenêtre modale. Le bouton 'Cancel' qui permet de fermer la fenêtre modale grâce à son attribut 'data-dismiss' et le bouton 'SignIn' associé à un bloc de callback qui envoie le message validate. La touche enter du clavier permet également d'activer le bouton 'SignIn' car c'est le seul dont l'attribut 'type' a la valeur 'submit' (ceci est réalisé par la méthode tbsSubmitButton).

```

TBAuthenticationComponent >> renderButtonsOn: html
    html tbsButton
        attributeAt: 'type' put: 'button';
        attributeAt: 'data-dismiss' put: 'modal';
        beDefault;
        value: 'Cancel'.
    html tbsSubmitButton
        bePrimary;
        callback: [ self validate ];
        value: 'SignIn'

```

Dans la méthode validate, nous envoyons simplement un message au composant principal en lui passant les identifiants rentrés par l'utilisateur.

```

TBAuthenticationComponent >> validate
    ^ component tryConnectionWithLogin: self account andPassword: self
    password

```

8.8 Intégration du composant d'identification

Pour intégrer notre composant d'identification, modifions le bouton 'Admin' de la barre d'en-tête (TBHeaderComponent) ainsi:

```

TBHeaderComponent >> renderButtonsOn: html
    self renderModalLoginButtonOn: html

TBHeaderComponent >> renderModalLoginButtonOn: html
    html render: (TBAuthenticationComponent from: component).
    html tbsNavbarButton
        tbsPullRight;
        attributeAt: 'data-target' put: '#myAuthDialog';
        attributeAt: 'data-toggle' put: 'modal';
        with: [
            html tbsGlyphIcon iconLock.
            html text: ' Login' ]

```

La méthode `renderModalLoginButtonOn:` commence par intégrer le code du composant `TBAuthenticationComponent` dans la page web (`render:`). Le composant étant instancié à chaque affichage, il n'a pas besoin d'être retourné par la méthode `children`. On ajoute également un bouton nommé 'Login' avec un pictogramme un pictogramme clé. Lorsque l'utilisateur clique sur ce bouton, la boîte modale ayant l'identifiant `myAuthDialog` est affichée.

En rechargeant la page de TinyBlog dans votre navigateur, nous voyons maintenant un bouton 'Login' dans l'en-tête permettant d'ouvrir la fenêtre modale comme illustré sur la figure 8-5.

8.9 Gestion naive des logins

Toutefois, si vous cliquez sur le bouton 'SignIn', une erreur se produit. En utilisant le debugger Pharo, on comprend qu'il faut définir la méthode `tryConnectionWithLogin:andPassword:` sur le composant `TBPostsListComponent` car c'est le message envoyé par le callback du bouton 'SignIn' de la fenêtre modale:

```

TBPostsListComponent >> tryConnectionWithLogin: login andPassword:
    password
    (login = 'admin' and: [ password = 'topsecret' ])
        ifTrue: [ self goToAdministrationView ]
        ifFalse: [ self loginErrorOccurred ]

```

Pour l'instant, le login et le mot de passe pour accéder à la partie administration sont directement stockés en dur dans le code de cette méthode ce qui n'est pas très bon.

8.10 Gestion des erreurs

Nous avons déjà défini la méthode `goToAdministrationView` précédemment. Ajoutons la méthode `loginErrorOccured` et un mécanisme pour afficher un message d'erreur lorsque l'utilisateur n'utilise pas les bons identifiants comme sur la figure 8-6.

Pour cela nous ajoutons une variable d'instance `showLoginError` qui représente le fait que nous devons afficher une erreur.

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

La méthode `loginErrorOccurred` spécifie qu'une erreur doit être affichée.

```
TBPostsListComponent >> loginErrorOccurred
  showLoginError := true
```

Nous ajoutons une méthode pour tester cet état.

```
TBPostsListComponent >> hasLoginError
  ^ showLoginError ifNil: [ false ]
```

Nous définissons aussi un message d'erreur.

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

Nous modifions la méthode `renderPostColumnOn:` afin de faire un traitement spécifique en cas d'erreur.

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

La méthode `renderLoginErrorMessageIfAnyOn:` affiche si nécessaire un message d'erreur. Elle repositionne la variable d'instance `showLoginError` pour ne que le message ne soit pas afficher indéfiniment.

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

8.11 Modélisation des administrateurs

Nous ne souhaitons pas stocker les identifiants administrateur du blog dans le code comme nous l'avons fait précédemment. Nous allons maintenant réviser cela et stocker ces identifiants dans le modèle.

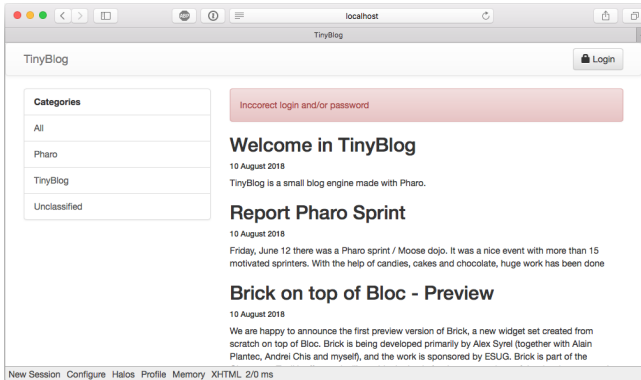


Figure 8-6 Message d'erreur en cas d'identifiants erronés.

Commençons par enrichir notre modèle de Tinyblog avec la notion d'administrateur. Ajoutons donc une nouvelle classe nommée `TBAdministrator` caractérisé par son pseudo, son login et son mot de passe.

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

```
TBAdministrator >> login
^ login
```

```
TBAdministrator >> login: anObject
login := anObject
```

```
TBAdministrator >> password
^ password
```

Notez que nous ne stockons pas le mot de passe administrateur en clair dans la variable d'instance `password` mais son hash en MD5.

```
TBAdministrator >> password: anObject
password := MD5 hashMessage: anObject
```

Nous définissons aussi une méthode de création.

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

Vous pouvez vérifier cela en inspectant l'expression suivante :

```
[luc := TBAdministrator login: 'luc' password: 'topsecret'.
```

8.12 Administrateur pour un blog

Un blog possède un administrateur qui peut s'identifier sur le blog afin administrer les posts qu'il contient. Ajoutons donc un champ `adminUser` et un accesseur en lecture dans la classe `TBBlog` afin d'y stocker l'administrateur du blog:

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

TBBlog >> administrator
  ^ adminUser
```

Nous définissons le login et password que nous utiliserons par défaut. Comme vous allez le voir plus loin, nous allons modifier les attributs de l'administrateur et ceux-ci seront sauves en même temps que le blog dans la base de données.

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

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

Maintenant nous nous pouvons créer une administrateur par défaut.

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

Lors de l'initialisation d'un blog ajoutons un administrateur par défaut.

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

8.13 Définir un administrateur

Il ne faut pas oublier de re-crée le blog ainsi:

```
[ TBBlog reset; createDemoPosts
```

Vous pouvez maintenant modifier le login et le mot de passe administrateur de votre blog ainsi:

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

Notez que sans rien faire, l'administrateur du blog a été sauvegardé par Voyage dans la base de données. En effet, la classe TBBlog étant une racine Voyage, tous ces attributs sont stockés dans la base automatiquement lors de l'envoi du message save.

Améliorations possibles

Etendre le modèle de l'application ainsi nécessite l'écriture de nouveaux tests unitaires. A vous de jouer!

8.14 Intégration du compte administrateur

Modifions maintenant la méthode tryConnectionWithLogin:andPassword: pour qu'elle utilise les identifiants de l'administrateur du blog courant. Notez que nous comparons les hash MD5 des mots de passe car nous ne stockons pas le mot de passe en clair dans le modèle.

```
TBPostsListComponent >> tryConnectionWithLogin: login andPassword:
    password
    (login = self blog administrator login and: [
        (MD5 hashMessage: password) = self blog administrator password
    ])
        ifTrue: [ self goToAdministrationView ]
        ifFalse: [ self loginErrorOccurred ]
```

8.15 Stocker l'administrateur courant en session

Actuellement, si l'administrateur du blog veut naviguer entre la partie privée et la partie publique de TinyBlog, il doit se reconnecter à chaque fois. Nous allons simplifier cela en stockant l'administrateur courant en session lors d'une connexion réussie.

Un objet session est attribué à chaque instance de l'application. Il permet de conserver principalement des informations qui sont partagées et accessible entre les composants. Nous stockerons donc l'administrateur courant en session et modifierons les composants pour afficher des boutons permettant une navigation simplifiée lorsque l'administrateur est connecté. Lorsqu'il se déconnecte explicitement ou que la session expire, nous supprimerons la session courante.

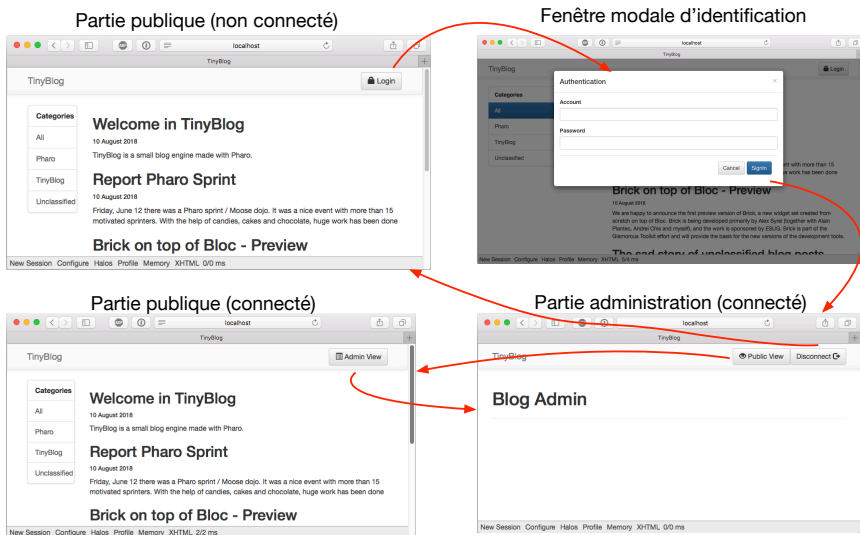


Figure 8-7 Navigation et identification dans TinyBlog.

La figure 8-7 illustre la navigation entre les pages que nous souhaitons mettre en place dans TinyBlog.

8.16 Définition et utilisation d'une classe session spécifique

Commençons par définir une nouvelle sous-classe de `WASession` nommée `TBSession` dans laquelle nous ajoutons une variable d'instance pour stocker l'administrateur connecté.

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

TBSession >> currentAdmin
    ^ currentAdmin

TBSession >> currentAdmin: anObject
    currentAdmin := anObject
```

Nous définissons une méthode `isLogged` qui nous permettra de savoir si l'administrateur est loggé.

```
TBSession >> isLogged
    ^ self currentAdmin notNil
```

Indiquons maintenant à Seaside qu'il doit utiliser l'objet `TBSession` comme objet de session courant pour l'application TinyBlog. Cette initialisation s'ef-

fectue dans la méthode initialize de la classe TBAApplicationRootComponent que l'on modifie ainsi:

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

Pensez à exécuter cette méthode via TBAApplicationRootComponent initialize avant de tester à nouveau l'application.

8.17 Stockage de l'administrateur courant en session

Lors d'une connexion réussie, nous ajoutons l'objet administrateur dans la session grâce à l'accesseur en écriture currentAdmin:. Notez que tout composant Seaside peut accéder à la session en cours en invoquant le message self session.

```
TBPostsListComponent >> tryConnectionWithLogin: login andPassword:
  password
  (login = self blog administrator login and: [
    (MD5 hashMessage: password) = self blog administrator password
  ])
    ifTrue: [
      self session currentAdmin: self blog administrator.
      self goToAdministrationView ]
    ifFalse: [ self loginErrorOccurred ]
```

8.18 Navigation simplifiée vers la partie administration

Pour mettre en place une navigation simplifiée, modifions l'en-tête pour afficher soit le bouton de connexion soit un bouton de navigation simple vers la partie administration sans étape de connexion si un administrateur est déjà connecté c'est-à-dire présent en session.

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

Vous pouvez tester dans votre navigateur en commençant une nouvelle session (bouton 'New Session' en bas à gauche). Une fois connecté, l'administra-

teur est ajouté en session. Remarquez que le bouton déconnexion ne fonctionne plus correctement car il n'invalide pas la session.

8.19 Déconnexion

Ajoutons une méthode `reset` sur notre objet session afin de supprimer l'administrateur courant, invalider la session courante et rediriger vers le point d'entrée de l'application.

```
TBSession >> reset
  currentAdmin := nil.
  self requestContext redirectTo: self application url.
  self unregister.
```

Modifions maintenant le bouton déconnexion de l'en-tête de la partie administration pour envoyer ce message `reset` à la session courante:

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

Le bouton 'Disconnect' fonctionne à nouveau correctement.

8.20 Navigation simplifiée vers la partie publique

Ajoutons maintenant un nouveau bouton dans l'en-tête de la partie administration pour revenir à la partie publique sans se déconnecter.

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

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

Vous pouvez maintenant tester la navigation dans votre application qui doit correspondre avec la représentation sur la figure 8-7.

8.21 Conclusion

Nous avons mis en place une gestion de l'identification pour TinyBlog. Cela comprend un composant réutilisable d'identification sous la forme d'une fenêtre modale. Nous avons également différencié les composants affichés lorsqu'un administrateur est connecté ou non. Enfin, nous avons utilisé la session pour faciliter la navigation d'un administrateur connecté jusqu'à sa déconnexion.

Nous voici prêts à définir la partie administrative de l'application ce qui est l'objet du chapitre suivant. Nous en profiterons pour vous montrer un aspect avancé qui permet la définition automatique de formulaires ou d'objets ayant de nombreux champs.

Améliorations possibles

A titre d'exercice, vous pouvez :

- afficher le login de l'administrateur dans l'en-tête lorsqu'il est connecté,
- ajouter la possibilité d'avoir plusieurs compte administrateurs chacun avec ses propres identifiants.

Interface web d'administration et génération automatique

Nous allons maintenant développer la partie administration de TinyBlog. Dans les chapitres précédents, nous avons défini des composants Seaside qui interagissent entre eux et où chaque composant est responsable son état et de son rendu graphique. Dans ce chapitre, nous voulons vous montrer que l'on peut aller encore plus loin et générer des composants Seaside à partir de la description d'objets en utilisant le framework Magritte.

La figure 9-1 montre une partie du résultat que nous allons obtenir. L'autre partie étant l'édition de bulletins.

La figure 9-2 montre un aperçu de l'architecture visée dans ce chapitre.

9.1 Décrire les données métiers avec Magritte

Magritte est une bibliothèque qui permet une fois les données décrites de générer diverses représentations ou opérations (telles des requêtes). Couplé avec Seaside, Magritte permet de générer des formulaires et des rapports. Le logiciel Quuve de la société Debris Publishing est un brillant exemple de la puissance de Magritte: tous les tableaux sont automatiquement générés (voir <http://www.pharo.org/success>). La validation des données est aussi définie au niveau de Magritte au lieu d'être dispersée dans le code de l'interface graphique. Ce chapitre ne montre pas cet aspect.

Un chapitre dans le livre sur Seaside (<http://book.seaside.st>) est disponible sur Magritte ainsi qu'un tutoriel en cours d'écriture sur <https://github.com/SquareBracketAssociates/Magritte>.

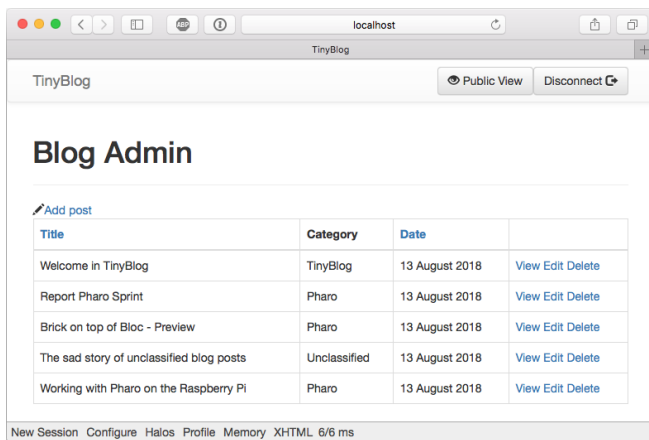


Figure 9-1 Gestion des bulletins.

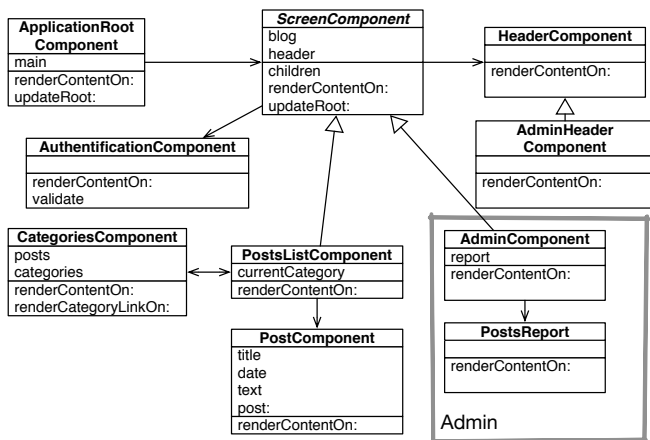


Figure 9-2 Composants pour l'administration.

Une description est un objet qui spécifie des informations sur des données de notre modèle comme son type, si une donnée est obligatoire, si elle doit être triée, ou quelle est sa valeur par défaut.

9.2 Description d'un bulletin

Commençons par décrire les cinq variables d'instance de l'objet `TBPost` à l'aide de Magritte. Ensuite, nous en tirerons avantage pour générer automatiquement des composants Seaside.

Les cinq méthodes suivantes sont dans le protocole 'magritte-descriptions' de la classe `TBPost`. Noter que le nom des méthodes n'est pas important mais que nous suivons une convention. C'est le pragma `<magritteDescription>` qui permet à Magritte d'identifier les descriptions.

Le titre d'un bulletin est une chaîne de caractères devant être obligatoirement complétée.

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

Le texte d'un bulletin est une chaîne de caractères multi-lignes devant être obligatoirement complété.

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

La catégorie d'un bulletin est une chaîne de caractères qui peut ne pas être renseignée. Dans ce cas, le post sera de toute manière rangé dans la catégorie 'Unclassified'.

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

La date de création d'un bulletin est importante car elle permet de définir l'ordre de tri pour l'affichage des posts. C'est donc une variable d'instance contenant obligatoirement une date.

```
TBPost >> descriptionDate
  <magritteDescription>
  ^ MAMDateDescription new
    accessor: #date;
    beRequired;
    yourself
```

La variable d'instance visible doit obligatoirement contenir une valeur booléenne.

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

Nous pourrions enrichir les descriptions pour qu'il ne soit pas possible de poster un bulletin ayant une date antérieure à celle du jour. Nous pourrions changer la description d'une catégorie pour que ses valeurs possibles soient définies par l'ensemble des catégories existantes. Tout cela permettrait de produire des interfaces plus complètes et toujours aussi simplement.

9.3 Création automatique de composant

Une fois un bulletin décrit, nous pouvons générer un composant Seaside en envoyant le message `asComponent` à une instance.

```
[ aTBPost asComponent
```

Nous allons voir comment utiliser cela dans la suite.

9.4 Mise en place d'un rapport des bulletins

Nous allons développer un nouveau composant qui sera utilisé par le composant `TBAdminComponent`. Le composant `TBPostReport` est un rapport qui contiendra tous les posts. Comme nous allons le voir, le rapport est automatiquement généré. Le rapport étant généré par Magritte sous la forme d'un composant Seaside, nous aurions pu n'avoir qu'un seul composant. Toutefois, nous pensons que distinguer le composant d'administration du rapport est une bonne chose pour l'évolution de la partie administration.

Le composant `PostsReport`

La liste des posts est affichée à l'aide d'un rapport généré dynamiquement par le framework Magritte. Nous utilisons ce framework pour réaliser les différentes fonctionnalités de la partie administration de TinyBlog (liste des posts, création, édition et suppression d'un post).

Pour rester modulaire, nous allons créer un composant Seaside pour cette tâche. Le composant `TBPostsReport` est une sousclasse de la classe `TBS-MagritteReport` qui gère les rapports avec Bootstrap.

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

Nous ajoutons une méthode de création qui prend en argument un blog et donc ses bulletins.

```
TBPostsReport class >> from: aBlog
  | allBlogs |
  allBlogs := aBlog allBlogPosts.
  ^ self rows: allBlogs description: allBlogs anyOne
    magritteDescription
```

9.5 Intégration de PostsReport dans AdminComponent

Révisons maintenant notre composant `TBAdminComponent` pour afficher ce rapport. On ajoute une variable d'instance `report` et ses accesseurs à la classe `TBAdminComponent`.

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

```
TBAdminComponent >> report
  ^ report
```

```
TBAdminComponent >> report: aReport
  report := aReport
```

Comme le rapport est un composant fils du composant admin nous n'oublions pas de redéfinir la méthode `children` comme suit. Notez que la collection contient à la fois les sous-composants définis dans la super-classe (le composant en-tête) et ceux dans la classe courante (le composant rapport).

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

Dans la méthode `initialize`, nousinstancions un rapport tout en lui fournissant accès aux données du blog.

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

Modifions le rendu de la partie administration afin d'afficher le rapport.

```

TBAdminComponent >> renderContentOn: html
    super renderContentOn: html.
    html tbsContainer: [
        html heading: 'Blog Admin'.
        html horizontalRule.
        html render: self report ]

```

Vous pouvez déjà tester dans votre navigateur.

9.6 Filtrer les colonnes

Par défaut, un rapport affiche l'intégralité des données présentes dans chaque post. Cependant certaines colonnes ne sont pas utiles. Il faut donc filtrer les colonnes. Nous ne retiendrons ici que le titre, la catégorie et la date de rédaction.

Nous ajoutons une méthode de classe pour la sélection des colonnes et modifions ensuite la méthode `from:` pour en tirer parti.

```

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

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

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

```

La figure 9-3 montre ce que vous devez obtenir dans votre navigateur.

9.7 Amélioration du rapport

Le rapport généré est brut. Il n'y a pas de titres sur les colonnes et l'ordre d'affichage des colonnes n'est pas fixé. Celui-ci peut varier d'une instance à une autre. Pour gérer cela, il suffit de modifier les descriptions Magritte pour chaque variable d'instance. Nous spécifions une priorité et un titre (message `label:`) comme suit :

```

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

```

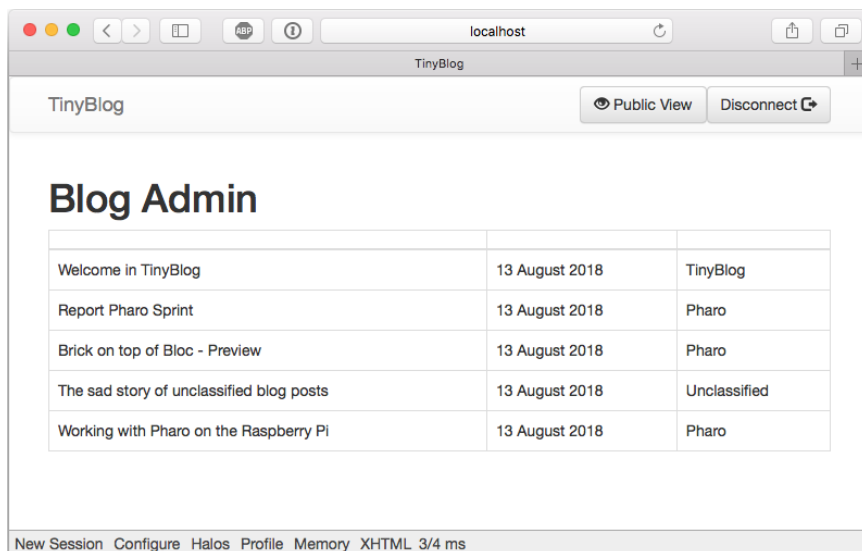


Figure 9-3 Rapport Magritte contenant les bulletins du blog.

```

        yourself

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

[ TBPPost >> descriptionCategory
  <magritteDescription>
    ^ MAMemoDescription new
      label: 'Category';
      priority: 300;
      accessor: #category;
      yourself

[ TBPPost >> descriptionDate
  <magritteDescription>
    ^ MAMemoDescription new
      label: 'Date';
      priority: 400;
      accessor: #date;
      beRequired;
      yourself

```

Title	Category	Date
Welcome in TinyBlog	TinyBlog	13 August 2018
Report Pharo Sprint	Pharo	13 August 2018
Brick on top of Bloc - Preview	Pharo	13 August 2018
The sad story of unclassified blog posts	Unclassified	13 August 2018
Working with Pharo on the Raspberry PI	Pharo	13 August 2018

Figure 9-4 Administration avec un rapport.

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

Vous devez obtenir la situation telle que représentée par la figure 9-4.

9.8 Administration des bulletins

Nous pouvons maintenant mettre en place un CRUD (Create Read Update Delete) permettant de gérer les bulletins. Pour cela, nous allons ajouter une colonne (instance `MACommandColumn`) au rapport qui regroupera les différentes opérations en utilisant `addCommandOn:`. Cette méthode permet de définir un lien qui déclenchera l'exécution d'une méthode de l'objet courant lorsqu'il sera cliqué grâce à un callback.

Ceci se fait lors de la création du rapport. En particulier nous donnons un accès au blog depuis le rapport.

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

```

TBSMagritteReport >> blog
  ^ blog

TBSMagritteReport >> blog: aTBBlog
  blog := aTBBlog

```

La méthode `from` : ajoute une nouvelle colonne au rapport. Elle regroupe les différentes opérations en utilisant `addCommandOn` :

```

TBPostsReport class >> from: aBlog
  | report blogPosts |
  blogPosts := aBlog allBlogPosts.
  report := self rows: blogPosts description: (self
    filteredDescriptionsFrom: blogPosts anyOne).
  report blog: aBlog.
  report addColumn: (MACCommandColumn new
    addCommandOn: report selector: #viewPost: text: 'View';
    yourself;
    addCommandOn: report selector: #editPost: text: 'Edit';
    yourself;
    addCommandOn: report selector: #deletePost: text: 'Delete';
    yourself).
  ^ report

```

Nous allons devoir définir les méthodes liées à chaque opérations dans une prochaine section.

Par ailleurs, cette méthode est un peu longue et elle ne permet pas de séparer la définition du rapport de l'ajout d'opérations sur les éléments. Une solution est de créer une méthode d'instance `addCommands` et de l'appeler explicitement. Faites cette transformation.

9.9 Gérer l'ajout d'un bulletin

L'ajout (`add`) est dissocié des bulletins et se trouvera donc juste avant le rapport. Etant donné qu'il fait partie du composant `TBPostsReport`, nous devons redéfinir la méthode `renderContentOn` : du composant `TBPostsReport` pour insérer le lien `add`.

```

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

```

Identifiez-vous à nouveau et vous devez obtenir la situation telle que représentée par la figure 9-5.

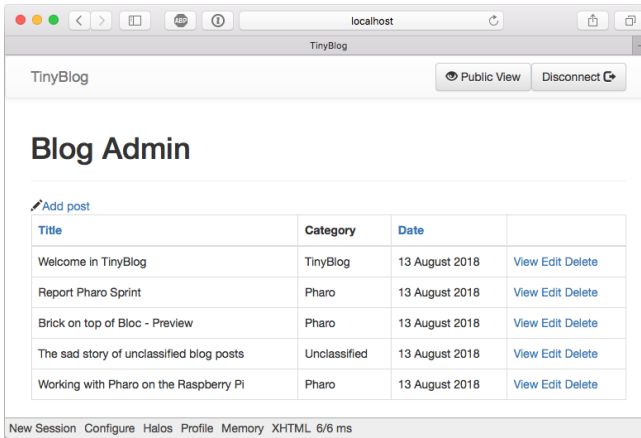


Figure 9-5 Rapport des bulletins avec des liens d'édition.

9.10 Implémentation des actions CRUD

A chaque action (Create/Read/Update/Delete) correspond une méthode de l'objet `TBPostsReport`. Nous allons maintenant les implémenter. Un formulaire personnalisé est construit en fonction de l'opération demandée (il n'est pas utile par exemple d'avoir un bouton "Sauver" alors que l'utilisateur veut simplement lire le post).

9.11 Ajouter un bulletin

Commençons par gérer l'ajout d'un bulletin. La méthode `renderAddPostForm` suivante illustre la puissance de Magritte pour générer des formulaires.

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

Ici, le message `asComponent`, envoyé à un objet métier instance de la classe `TBPost`, crée directement un composant `Seaside`. Nous ajoutons une décoration à ce composant `Seaside` afin de gérer ok/cancel.

La méthode `addPost` pour sa part, affiche le composant rendu par la méthode `renderAddPostForm`: et lorsque qu'un nouveau post est créé, elle l'ajoute au blog. La méthode `writeBlogPost`: sauve les changements.

The screenshot shows the 'Blog Admin' interface of 'TinyBlog'. At the top right, there are buttons for 'Public View' and 'Disconnect'. The main form is titled 'Blog Admin' and contains the following fields:

- Title:** A text input field containing 'Soon new TB version'.
- Text:** A large text area for the post content.
- Category:** A dropdown menu currently showing 'Unclassified'.
- Date:** A date input field showing '20 August 2018' with a 'Choose' button next to it.
- Visible:** A checkbox labeled 'Visible' which is currently unchecked.
- Buttons:** 'Add post' (in blue) and 'Cancel' (in grey) buttons at the bottom.

Figure 9-6 Affichage rudimentaire d'un bulletin.

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

On voit une fois encore l'utilisation du message `call:` pour donner la main à un composant. Le lien pour ajouter un bulletin permet maintenant d'afficher un formulaire de création que nous rendrons plus présentable (Voir figure 9-6).

Afficher un bulletin

Pour afficher un bulletin en lecture nous définissons deux méthodes similaires aux précédentes. Notez que nous utilisons l'expression `readonly: true` pour indiquer que le formulaire n'est pas éditabile.

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

Voir un bulletin ne nécessite pas d'action supplémentaire que d'afficher le composant.

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

Editer un bulletin

Pour éditer un bulletin nous utilisons la même approche.

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

Maintenant la méthode `editPost:` récupère la valeur du message `call:` et sauve les changements apportés.

```
TBPostsReport >> editPost: aPost
  | post |
  post := self call: (self renderEditPostForm: aPost).
  post ifNotNil: [ blog save ]
```

Supprimer un bulletin

Il nous faut maintenant ajouter la méthode `removeBlogPost:` à la classe `TBBlog:`

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

ainsi qu'un test unitaire :

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

Pour éviter une opération accidentelle, nous utilisons une boîte modale pour que l'utilisateur confirme la suppression du post. Une fois le post effacé, la liste des posts gérés par le composant `TBPostsReport` est actualisée et le rapport est rafraîchi.

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

9.12 Gérer le rafraîchissement des données

Les méthodes `addPost:` et `deletePost:` font bien leur travail mais les données à l'écran ne sont pas mises à jour. Il faut donc rafraîchir la liste des bulletins en utilisant l'expression `self refresh`.


```

[ TBPpostsReport >> refreshReport
  self rows: blog allBlogPosts.
  self refresh.

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

[ TBPpostsReport >> deletePost: aPost
  (self confirm: 'Do you want remove this post ?')
    ifTrue: [ blog removeBlogPost: aPost.
              self refreshReport ]

```

Le rapport est maintenant fonctionnel et gère même les contraintes de saisie c'est-à-dire que le formulaire assure par exemple que les champs déclarés comme obligatoire dans les descriptions Magritte sont bien renseignés.

9.13 Amélioration de l'apparence des formulaires

Pour tirer parti de Bootstrap, nous allons modifier les définitions Magritte. Tout d'abord, spécifions que le rendu du rapport doit se baser sur Bootstrap.

Un container en Magritte est l'élément qui va contenir les composants créer à partir des descriptions.

```

[ TBPpost >> descriptionContainer
  <magritteContainer>
  ^ super descriptionContainer
    componentRenderer: TBSMagritteFormRenderer;
    yourself

```

Nous pouvons maintenant nous occuper des différents champs de saisie et améliorer leur apparence.

```

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

```

The screenshot shows a web browser window titled 'TinyBlog' with the address 'localhost'. The page content is titled 'Blog Admin' and includes a 'Public View' button and a 'Disconnect' button. The main form has the following fields:

- Title:** A text input field containing 'Welcome in TinyBlog'. Below it is the text 'Please enter a title'.
- Text:** A text area containing 'TinyBlog is a small blog engine made with Pharo.'. Below it is the text 'Please enter a text'.
- Category:** A text input field containing 'TinyBlog'. Below it is the text 'Unclassified if empty'.
- Date:** A date picker showing '13 August 2018' and a 'Choose' button.
- Visible:** A checkbox that is checked, with the label 'Visible'.
- Back:** A blue button at the bottom of the form.

The browser's status bar at the bottom indicates 'New Session Configure Halos Profile Memory XHTML 0/2 ms'.

Figure 9-7 Formulaire d'ajout d'un post avec Bootstrap.

```

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

TBPPost >> descriptionCategory
<magritteDescription>
^ MAMStringDescription new
    label: 'Category';
    priority: 300;
    accessor: #category;
    comment: 'Unclassified if empty';
    componentClass: TBSMagritteTextInputComponent;
    yourself

```

```

TBlog >> descriptionVisible
  <magritteDescription>
    ^ MABooleanDescription new
      checkboxLabel: 'Visible';
      priority: 500;
      accessor: #visible;
      componentClass: TBSMagritteCheckboxComponent;
      beRequired;
      yourself

```

Grâce à ces nouvelles descriptions Magritte, les formulaires générés sous la forme de composants Seaside utilisent Bootstrap. Par exemple, formulaire d'édition d'un post doit maintenant ressembler à celui de la figure 9-7.

9.14 Conclusion

Nous avons mis en place la partie administration de TinyBlog sous la forme d'un rapport des bulletins contenus dans le blog courant. Nous avons également ajouté des liens permettant une gestion CRUD de chaque bulletin. Nous avons réalisé tout cela en utilisant Magritte. En effet, nous avons ajouté des descriptions sur les bulletins et généré des composants Seaside (des formulaires) à partir de ces descriptions.



Déploiement de TinyBlog

Dans ce chapitre nous vous montrons comment déployer votre application. Nous montrons en particulier un déploiement sur un serveur cloud.

10.1 Déployer dans le cloud

Maintenant que vous avez développé votre application web TinyBlog, nous allons voir comment la déployer sur un serveur dans le cloud. Si vous souhaitez déployer votre application sur un serveur que vous administrez, nous conseillons la lecture du dernier chapitre du livre "Enterprise Pharo: a Web Perspective" (<http://books.pharo.org>). Dans la suite, nous détaillons une solution plus simple fournie par PharoCloud.

10.2 Hébergement sur PharoCloud

PharoCloud est un hébergeur dédié aux applications Pharo et qui offre la possibilité de tester gratuitement ses services (ephemeric cloud subscription).

Préparer son compte PharoCloud :

- créer un compte sur <http://pharocloud.com>
- activer son compte
- se connecter
- activer "Ephemeric Cloud" afin d'obtenir un identifiant (**API User ID**) et un mot de passe (**API Auth Token**)

- cliquer sur "Open Cloud Client" et identifiez-vous avec les identifiants ci-dessus
- une fois connecté, vous devez obtenir une page web permettant d'uploader un fichier archive (zip) contenant un fichier image Pharo et son fichier changes

10.3 Préparation de l'image Pharo pour PharoCloud

Repartir d'une image vierge

Télécharger une image PharoWeb neuve <http://files.pharo.org/mooc/image/PharoWeb-50.zip>. Lancer cette image avec la VM correspondante et nous allons maintenant la configurer.

Configuration de Seaside

Commençons par configurer Seaside en enlevant les applications de démonstration et les outils de développement :

```
"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'.
```

Chargement de TinyBlog

Chargeons maintenant l'application TinyBlog. Pour charger l'application corrigée vous pouvez utiliser :

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

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

Mais vous pouvez aussi charger **votre** application TinyBlog depuis votre dépôt sur Smalltalkhub. Par exemple :

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

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

TinyBlog comme application par défaut

Indiquons maintenant à Seaside que l'application par défaut est TinyBlog et lançons le serveur HTTP.

```
"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 your image

Sauvegarder maintenant votre image Pharo (Menu World > save) et tester en local dans votre navigateur via l'URL <http://localhost:8080>.

10.4 Déploiement manuel sur Ephemeric Cloud de PharoCloud

Créer une archive (fichier zip) contenant les deux fichiers : `PharoWeb.image` et `PharoWeb.changes`.

Glisser/déposer ce fichier zip sur le Ephemeric Cloud de Pharo cloud et activer cette image (bouton play) comme indiqué sur la figure 10-1.

En cliquant sur l'URL publique fournie vous voir s'afficher votre application TinyBlog comme sur la figure 10-2.

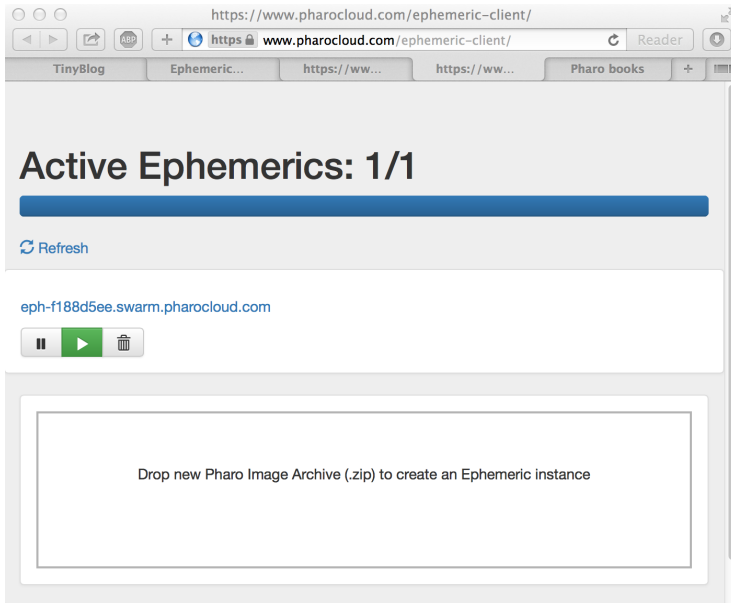


Figure 10-1 Administration des images Pharo sur Ephemeric Cloud.

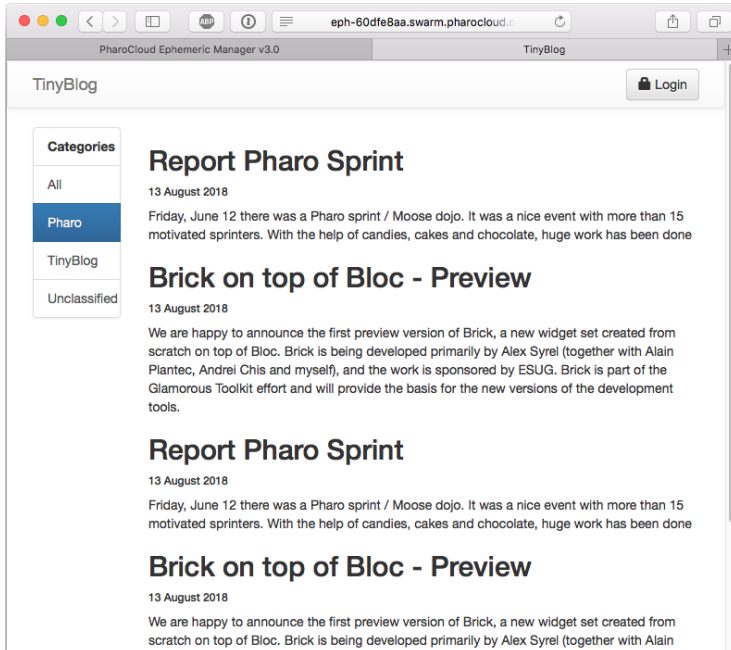


Figure 10-2 Votre application TinyBlog sur PharoCloud.

10.5 Déploiement automatique sur Ephemeric Cloud de PharoCloud

Plutôt que faire une archive zip et utiliser son navigateur, la documentation de PharoCloud (<http://docs.swarm.pharocloud.com/>) indique comment déployer directement son image Pharo en exécutant le code suivant (attention, c'est un peu long le temps que l'image soit transférée sur PharoCloud) :

```
|client EPHUSER EPHTOKEN|

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

ephUser := '<REST API UserID>'.
ephToken := '<REST API Token>'.
client := EphemericCloudClient userID: EPHUSER authToken: EPHTOKEN.
  (client publishSelfAs: 'glimpse')
    ifTrue:[ZnZincServerAdaptor startOn: 8080]
    ifFalse: [ client lastPublishedInstance hostname ]
```

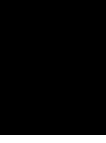
10.6 A propos de dépendances

Les bonnes pratiques lors de développements en Pharo sont de spécifier clairement les dépendances sur les packages utilisés afin d'avoir une reproductibilité complète d'un projet. Une telle reproductibilité permet alors l'utilisation de serveur de construction tels que Travis ou Jenkins. Pour cela, une configuration (une classe spéciale) définit d'une part l'architecture du projet (dépendances et packages du projet) et les versions des packages versionnés. L'image PharoWeb est automatiquement construite à partir d'une telle configuration.

Dans le cadre de ce projet, nous n'abordons pas ce point plus avancé. Un chapitre entier est consacré à l'expression de configurations dans le livre intitulé "Deep Into Pharo" (cf. <http://books.pharo.org>).

Part II

Eléments optionnels



Charger le code des chapitres

Ce chapitre contient les expressions permettant de charger le code décrit dans chacun des chapitres. Ces expressions doivent être exécutées dans une image PharoWeb 5 (<http://files.pharo.org/mooc/image/PharoWeb-5.0.zip>) ou PharoWeb 6.1 (<http://files.pharo.org/mooc/image/PharoWeb-61.zip>). Ensuite, si vous commencer le chapitre 4 par exemple, vous pouvez charger tout le code des chapitres précédents (1, 2 et 3) en suivant la procédure décrite dans la section 'Chapitre 4' ci-après.

Bien évidemment, nous vous conseillons de faire votre propre code mais cela vous permettra de ne pas rester bloqué le cas échéant.

11.1 Chapitre 3 : Extension du modèle et tests unitaires

Vous pouvez charger la correction du chapitre 2 en exécutant le code suivant :

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

Après le chargement d'un package, il est recommandé d'exécuter les tests unitaires qu'il contient afin de vérifier le bon fonctionnement du code chargé. Pour cela, vous pouvez lancer l'outil TestRunner (World menu > Test Runner), chercher le package TinyBlog-Tests et lancer tous les tests unitaires de la classe TBBlogTest en cliquant sur le bouton "Run Selected". Tous les tests doivent être verts. Une alternative est de presser l'icone verte qui se situe à coté de la class TBBlogTest.

11.2 Chapitre 4 : Persistance des données de TinyBlog avec Voyage et Mongo

Vous pouvez charger la correction du chapitre 3 en exécutant le code suivant:

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

Ouvrez maintenant un browser de code pour regarder le code des classes TBBlog et TBBlogTest et compléter votre propre code si nécessaire. Avant de poursuivre, n'oubliez pas de committer une nouvelle version dans votre dépôt si vous avez modifié votre application.

11.3 Chapitre 5 : Commencer avec Seaside

Vous pouvez charger la correction du chapitre 4 en exécutant le code suivant:

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

Exécutez les tests.

Pour tester l'application, vous devez lancer le serveur HTTP pour Seaside:

```
[ ZnZincServerAdaptor startOn: 8080.
```

Ouvrez votre browser sur <http://localhost:8080/TinyBlog>

Si vous avez besoin de créer quelques posts initiaux:

```
[ TBBlog reset ; createDemoPosts
```

11.4 Chapitre 6 : Des composants web pour TinyBlog

Vous pouvez charger la correction des chapitres précédents en exécutant le code suivant:

```
[ Metacello new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  version: #chapter5solution;
  configuration: 'TinyBlog';
  load
```

Pour tester le code, vous devez lancer le serveur HTTP pour Seaside:

```
[ ZnZincServerAdaptor startOn: 8080.
```

Ouvrez votre browser sur <http://localhost:8080/TinyBlog>

Si vous avez besoin de créer quelques posts initiaux:

```
[ TBBlog reset ; createDemoPosts
```

11.5 Chapitre 7 : Gestion des catégories

Vous pouvez charger la correction des chapitres précédents en exécutant le code suivant:

```
[ Metacello new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  version: #chapter6solution;
  configuration: 'TinyBlog';
  load
```

Même process que le précédent.

11.6 Chapitre 8 : Authentification et Session

Vous pouvez charger l'application TinyBlog avec la partie publique en exécutant :

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

Pour tester le code, vous devez lancer le serveur HTTP pour Seaside:

```
[ ZnZincServerAdaptor startOn: 8080.
```

11.7 Chapitre 9 : Interface Web d'administration et génération automatique

Vous pouvez charger l'application TinyBlog avec l'authentification en exécutant :

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

11.8 Chapitre 10 : Déploiement de TinyBlog

Vous pouvez charger l'application TinyBlog contenant la partie administration en exécutant :

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

Vous devez également créer un compte sur PharoCloud, reporter vos identifiants dans le code ci-dessous :

```
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 := #EphemicCloudClient asClass userID: ephUser authToken:
    ephToken.
(client publishSelfAs: 'TinyBlog')
    ifTrue:[ ZnZincServerAdaptor startOn: 8080 ]
    ifFalse: [ client lastPublishedInstance hostname ]
```

Si vous exécutez ce code (cela prend un peu de temps), votre image Pharo courante sera automatiquement envoyée sur votre espace PharoCloud.



Sauver votre code

Lorsque vous sauvez l'image Pharo (clic gauche sur le fond de Pharo pour accéder au menu world et sélectionner 'save'), celle-ci contient tous les objets du système et donc les classes elles-même. Cette solution est pratique mais peu pérenne. Nous allons vous montrer comment les pharoers sauvent leur code sous forme de packages sur un serveur dédié à l'aide de Monticello: le gestionnaire de versions de Pharo.

Notez que des videos sont disponibles dans le Mooc Pharo qui montrent la procédure pour sauver du code: <http://mooc.pharo.org> et en particulier la video de la semaine 1 qui montre comment développer et sauver le code d'une application compteur: http://rmod-pharo-mooc.lille.inria.fr/MOOC/Videos/W1/C019-W1S-Videos-Redo-Counter-Traditional-FR-V3-HD_720p_4Mbs.m4v.

Créer un dépôt de code

Il existent plusieurs serveurs en ligne permettant d'héberger vos dépôts de code gratuitement comme Smalltalkhub <http://smalltalkhub.com> ou SS3 <http://ss3.gemstone.com>.

- Créer un compte sur le site <http://smalltalkhub.com/>.
- Se connecter au site.
- Créer un projet nommé "TinyBlog" (si vous rencontrez des problèmes de connexion car le site est en version beta, essayez avec un autre navigateur ; si les problèmes persistent utilisez <http://ss3.gemstone.com>).

Sauver votre package

- Dans Pharo, ouvrez l'outil Monticello Browser via le menu principal (clic gauche sur le fond de Pharo).
- Ajoutez un dépôt (repository) de type SmalltalkHub ou HTTP pour `http://ss3.gemstone.com`.
- Sélectionnez ce dépôt et dans son menu contextuel (clic droit), sélectionnez l'item 'Add to package...' pour ajouter ce dépôt au package TinyBlog.
- Sélectionnez maintenant votre package et pressez le bouton 'Save'.
- Indiquez une description pour votre commit et sauvez. Votre code vient d'être envoyé sur le serveur.

Le code de votre application TinyBlog est maintenant sauvegardé dans votre dépôt sur Smalltalkhub. Il est donc maintenant possible de charger votre code dans une nouvelle image Pharo. Pour rappel, dans le cadre de ce tutoriel, nous vous suggérons de toujours utiliser l'image avec tous les packages web chargés que nous avons mentionné dans le premier chapitre. Cela vous permettra de pouvoir recharger votre package sans devoir vous soucier des dépendances sur d'autres packages.