



TinyBlog: Créer votre Première Web App avec Pharo

Olivier Auverlot, Stéphane Ducasse et Luc
Fabresse

March 10, 2018
master @ 34d2b4f

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	iv
1 A propos de ce livre	1
1.1 Structure	1
1.2 Installation de Pharo	1
1.3 Règles de nommage	2
1.4 Ressources	2
I Tutoriel de base	
2 L'application TinyBlog : présentation et modèle	5
2.1 La classe TPost	5
2.2 Gérer la visibilité d'un post	6
2.3 Initialisation	7
2.4 Méthodes de création	7
2.5 Création de posts	7
2.6 Interrogation d'un post	7
2.7 Sauvegarder votre code	8
2.8 A propos de dépendances	9
2.9 Conclusion	9
3 L'application TinyBlog : extension du modèle et tests unitaires	11
3.1 La classe TBlog	11
3.2 Un seul blog	11
3.3 Tester les règles métiers	12
3.4 Un premier test	13
3.5 Données de test	14
3.6 Futures évolutions	15
3.7 Conclusion	15
4 Persistance des données de TinyBlog avec Voyage et Mongo	17
4.1 Configurer Voyage pour sauvegarder des objets TBlog	17
4.2 Sauvegarde d'un blog	18
4.3 Utilisation de la base	18
4.4 Si nous devons sauvegarder les posts [Optionnel]	19
4.5 Tester avec une base Mongo [Optionnel]	20
4.6 Installation de Mongo	20
4.7 Connexion à un serveur local	20
4.8 Conclusion	21
5 Une interface Web en Seaside pour TinyBlog	23
5.1 Démarrer Seaside	23
5.2 Point d'entrée de l'application	24
5.3 Premier rendu simple	25
5.4 Composants visuels pour TinyBlog	26
5.5 Bootstrap for Seaside	28

5.6	Définition du composant TBHeaderComponent	28
5.7	Utilisation du composant header	29
5.8	Relation composite-composant	29
5.9	Utilisation du composant Screen	29
5.10	Liste des posts	31
5.11	Le composant Post	31
5.12	Afficher les posts	33
5.13	Débugger les erreurs	33
5.14	Affichage de la liste des posts avec Bootstrap	34
5.15	Relation composite composants	34
5.16	Affichage des posts par catégorie	35
5.17	Rendu des catégories	36
5.18	Mise à jour des Posts	37
5.19	Look et agencement	37
5.20	Notre solution: Plein de petites méthodes	39
5.21	Conclusion	40
6	Interface Web d'administration pour TinyBlog	43
6.1	Administration de TinyBlog	43
6.2	Composant d'identification	43
6.3	Intégration de l'authentification	46
6.4	Décrire les données métiers avec Magritte	46
6.5	Administration des posts	48
6.6	Le composant PostsReport	48
6.7	Amélioration des rapports	50
6.8	Gestion des posts	51
6.9	Implémentation des actions CRUD	52
6.10	Gérer le problème du rafraîchissement des données	54
6.11	Amélioration de l'apparence du formulaire	54
6.12	Gestion de session	55
6.13	Améliorations possibles	57
7	Déploiement de TinyBlog	59
7.1	Déployer dans le cloud	59
7.2	Hébergement sur PharoCloud	59
7.3	Préparation de l'image Pharo pour PharoCloud	59
7.4	Déploiement sur Ephemeric Cloud de PharoCloud	60
II	Éléments optionnels	
8	Construire une interface Web avec Teapot pour TinyBlog	65
8.1	Tester votre application	66
8.2	Afficher la liste des posts	67
8.3	Détails d'un Post	68
8.4	Amélioration possibles	69
9	Une interface REST pour TinyBlog	71
9.1	Notions de base sur REST	71
9.2	Définir un filtre REST	71
9.3	Obtenir la liste des posts	72
9.4	Créer des Services	73
9.5	Construire une réponse	74
9.6	Implémenter le code métier du service listAll	75
9.7	Utiliser un service REST	75
9.8	Recherche d'un Post	76
9.9	Chercher selon une période	77

9.10	Ajouter un post	78
9.11	Amélioration possibles	79
10	Maîtriser les feuilles de styles avec RenoirSt	81
10.1	Installation de RenoirSt	81
10.2	Principe de fonctionnement	81
10.3	Intégrer une feuille de style dans TinyBlog	82
10.4	Améliorer le visuel	82
10.5	Faciliter la maintenance	83
10.6	Conclusion	85
11	Utiliser des modèles de mise en page avec Mustache	87
11.1	Ajouter un bas de page	87
11.2	Ajouter du contenu dans le bas de page	89
11.3	Utiliser de texte statique	89
11.4	Utiliser du texte généré dynamiquement	89
11.5	Conclusion	89
12	Exportation de données	91
12.1	Exporter un article en PDF	91
12.2	Exportation des posts au format CSV	93
12.3	Ajouter l'option d'exportation	93
12.4	Implémentation de la classe TBPostsCSVExport	94
12.5	Exportation des posts au format XML	94
12.6	Génération des données XML	95
12.7	Amélioration possibles	96
13	Charger le code des chapitres	97
13.1	Chapitre : Modèle	97
13.2	Chapitre : Extension du modèle et tests unitaires	97
13.3	Chapitre : Persistance des données de TinyBlog avec Voyage et Mongo	97
13.4	Chapitre : Construction d'une interface Web en Seaside pour TinyBlog	98
13.5	Chapitre : Interface Web d'administration pour TinyBlog	98
13.6	Chapitre : Une Interface Web avec Teapot pour TinyBlog	98

Illustrations

1-1	L'application TinyBlog application	2
2-1	L'application TinyBlog.	6
2-2	Inspecteur sur une instance de TBPPost.	8
5-1	Lancer le serveur.	23
5-2	Vérification que Seaside fonctionne.	24
5-3	TinyBlog est bien enregistrée.	25
5-4	Une page quasi vide mais servie par Seaside.	25
5-5	Les composants composant l'application TinyBlog.	26
5-6	Accès à la bibliothèque Bootstrap.	27
5-7	Comprendre un élément et son code en Bootstrap.	28
5-8	TinyBlog avec une barre de navigation.	30
5-9	Le composant ApplicationRootComponent utilise de manière temporaire le composant ScreenComponent qui a un HeaderComponent.	30
5-10	Le composant ApplicationRootComponent utilise le composant PostsListComponent.	31
5-11	TinyBlog avec une liste de posts plutôt élémentaire.	32
5-12	Ajout du composant Post.	32
5-13	TinyBlog avec une liste de posts.	34
5-14	TinyBlog avec une liste de posts élémentaire.	35
5-15	Ajout du composant Categories.	36
5-16	Catégories afin de sélectionner les posts.	38
5-17	Avec un meilleur agencement.	39
5-18	Final TinyBlog Public UI.	40
6-1	Authentification avec un meilleur agencement.	44
6-2	Un composant d'administration vide.	49
6-3	Administration avec un rapport.	51
6-4	Ajout d'un post.	52
6-5	Ajout d'un post.	53
6-6	Formulaire d'ajout d'un post avec Bootstrap.	56
7-1	Administration des images Pharo sur Ephemer Cloud.	61
7-2	Votre application TinyBlog sur PharoCloud.	61
8-1	Une première page servie par notre application.	66
8-2	Afficher la liste des titres de posts.	67
10-1	Amélioration de l'apparence de TinyBlog	84
11-1	Le bas de page	88
11-2	Le bas de page	90
12-1	Chaque post peut être exporté en PDF	92

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 (voir l'état final de l'application dans la Figure 1-1). 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. Notre idée est que par la suite vous pourrez réutiliser cette infrastructure pour de multiples autres applications.

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/Seaside/Mongo.

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

1.2 Installation de Pharo

Dans ce tutoriel, nous supposons que vous utilisez Pharo 6.0 (<http://pharo.org/download>) avec une image dans laquelle ont été chargés des bibliothèques et des frameworks spécifiques pour le développement d'applications Web: Seaside, Magritte, Bootstrap, Voyage, VoyageMongo, ...

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-60.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

```
Metacello new
  baseline: 'PharoWebApp';
  repository: 'github://pharo-project/app-images';
  load
```

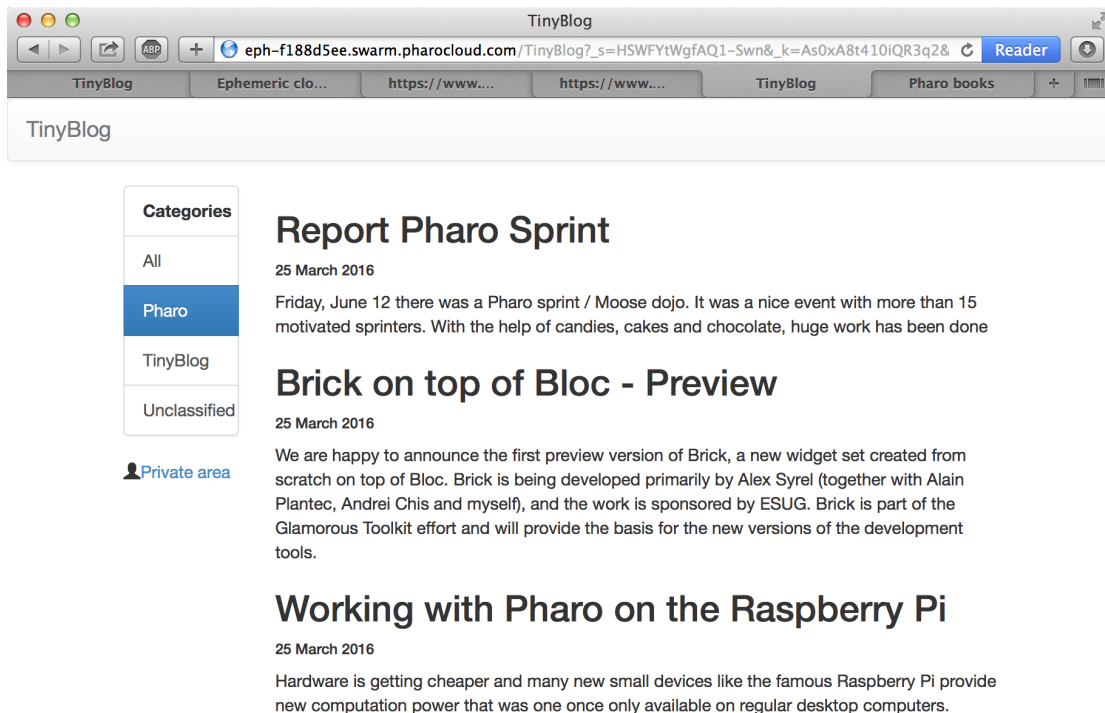


Figure 1-1 L'application TinyBlog application

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 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 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.
- <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

L'application TinyBlog : présentation et modèle

Tout au long de ce livre, nous allons vous guider pour développer et enrichir une application Web, nommée TinyBlog, pour gérer un ou plusieurs blogs (voir la Figure 2-1). 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.

2.1 La classe TBPPost

Le modèle de TinyBlog est extrêmement simple. Nous commençons ici par la classe TBPPost définie ainsi:

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

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

Variable	Signification
title	Titre du post
text	Texte du post
date	Date de rédaction
category	Rubrique contenant le post
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]
```

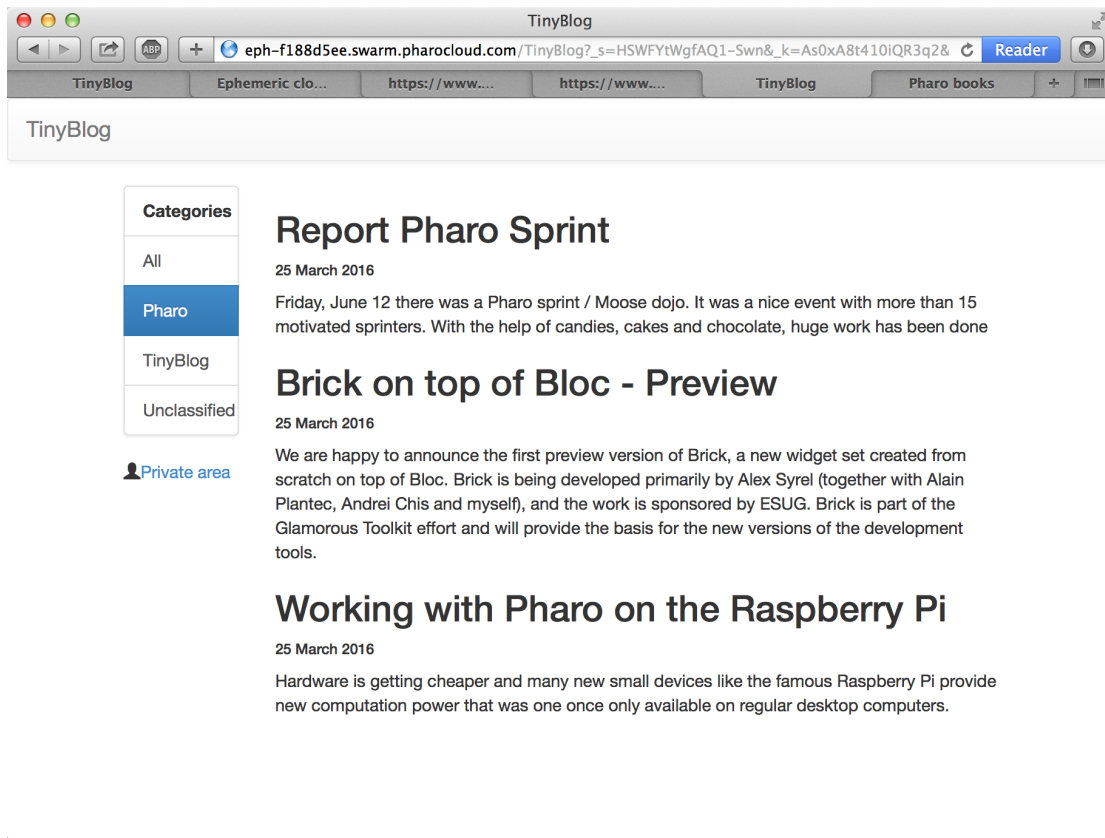


Figure 2-1 L'application TinyBlog.

```
[ 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 >> beNotVisible
  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 ce qui permet de rédiger des brouillons et de publier lorsque le post est terminé. Un post 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 beNotVisible
```

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`. Proposez une solution.

2.4 Méthodes de création

Coté classe, on définit des méthodes (dans la catégorie `'instance creation'`) pour faciliter la création de post appartenant ou pas à une catégorie.

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

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

2.5 Création de posts

Vous pouvez maintenant créer des posts. Ouvrez l'outil Playground et recopiez l'expression suivante :

```
TBPost
  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 Interrogation d'un post

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

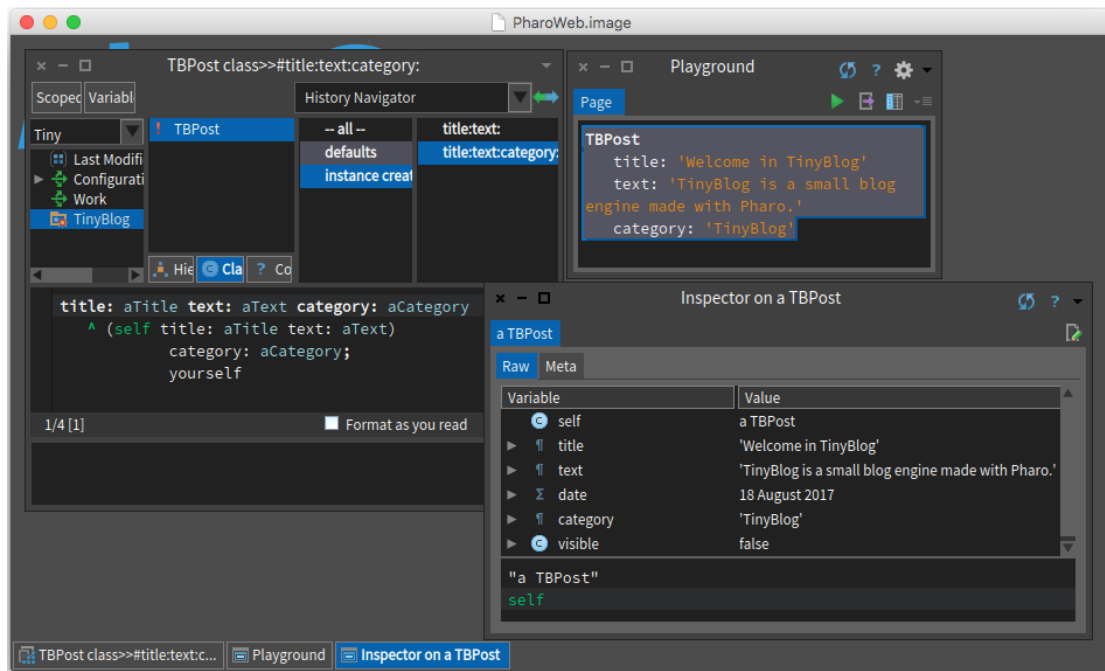


Figure 2-2 Inspecteur sur une instance de TBPost.

```
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 code d'une méthode. Proposer une solution.

2.7 Sauvegarder 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-mêmes. 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 vidéos sont disponibles dans le Mooc Pharo qui montrent la procédure pour sauver du code: <http://mooc.pharo.org> et en particulier la vidéo de la semaine 1 qui montre comment développer et sauver le code d'une application compteur: <http://W1/C019-W1S-Videos-Redo-Counter-Traditional-FR-v4.mp4>.

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 du monde (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.

2.8 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 tel 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.

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

2.9 Conclusion

Nous avons développé une première partie du modèle et appris à sauver notre code en utilisant Monticello.

L'application 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 les vidéos du Mooc de la second semaine et lire le livre *Learning Object-Oriented Programming, Design with TDD in Pharo* disponible a <http://books.pharo.org>.

3.1 La classe TBBlog

Nous allons développer classe TBBlog qui contient des posts, en écrivant des tests puis en les implémentant.

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

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

```
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. 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.

```
[ TBBlog class
  instanceVariableNames: 'uniqueInstance'

TBBlog class >> reset
  uniqueInstance := nil

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

Quand la classe est chargée, le singleton est réinitialisé.

```
[ 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 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. 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.

```
[ 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 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:` et `removeAllPosts`. Ajoutons-les.

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

TBBlog >> writeBlogPost: aPost
  "Write the blog post in database"
  posts add: aPost

TBBlog >> size
  ^ posts size
```

Le test précédent doit maintenant passer.

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
```

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
```

Obtenir tous les posts visibles

Test unitaire:

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

Code métier ajouté:

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

Obtenir tous les posts d'une catégorie

```

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

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

```

Obtenir tous les posts visibles d'une catégorie

```

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

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

```

Vérifier la gestion des posts non classés

```

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

```

Obtenir la liste des catégories

```

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

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

```

3.5 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: ((TBPost title: 'Brick on top of Bloc - Preview' text: 'We are
happy to announce the first preview version of Brick, a new widget set created from
scratch on top of Bloc. Brick is being developed primarily by Alex Syrel (together
with Alain Plantec, Andrei Chis and myself), and the work is sponsored by ESUG.
Brick is part of the Glamorous Toolkit effort and will provide the basis for the
new versions of the development tools.' category: 'Pharo') visible: true);
        writeBlogPost: ((TBPost title: 'The sad story of unclassified blog posts' text:
'So sad that I can read this.') visible: true);
        writeBlogPost: ((TBPost 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.

3.6 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.

3.7 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.

Persistence 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. Dans ce chapitre, nous voulons vous présenter une troisième option : la sauvegarde dans une base de données documents telle que Mongo. Voyage permet de sauver les objets dans une base de données Mongo. Voyage est un framework qui propose une API unifiée permettant d'accéder à différentes bases de données documents comme Mongo ou UnQLite.

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.

4.1 Configurer Voyage pour sauvegarder des objets TBBlog

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 données ce qui nous permet de ne pas avoir à préciser avec laquelle nous travaillons.

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

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

Ici nous n'avons pas besoin de stocker la base dans une variable d'instance car nous n'avons qu'une seule base de données en mode singleton.

Nous redéfinissons les méthodes de classe `reset` et `initialize` pour nous assurer que l'endroit où la base est sauvee est réinitialisé lorsque l'on charge le code.

La méthode `reset` réinitialise la base.

```
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 sauve notre blog et que le serveur s'éteigne par accident ou que nous rechargions 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 sauvee.

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 sauve 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 sauve le nouvel état d'un blog.

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

4.3 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 implémenté 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 queries sont plus pertinentes quand on a plus d'objets mais nous ferions exactement les mêmes.

4.4 Si nous devons sauvegarder les posts [Optionnel]

Cette section n'est pas à implémenter et elle est juste donnée à titre d'exemple. 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 souspartie d'un blog.

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. Par contre, 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.

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).

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 `TBBlog` 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.

4.5 Tester avec une base Mongo [Optionnel]

Nous allons maintenant montrer comment utiliser une base Mongo externe à Pharo. Dans le cadre de ce tutorial, 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.

4.6 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. Nous ne détaillons pas ici comment installer Mongo sur votre système, référez-vous à la documentation Mongo.

Note Le serveur Mongo ne doit pas utiliser d'authentification car la nouvelle méthode de chiffrement SCRAM utilisée par MongoDB 3.0 n'est actuellement pas supportée par Voyage.

Cependant vous devez créer une base de données nommée 'tinyblog'. Lorsque, vous avez installé et lancé un serveur Mongo localement, vous pouvez y accéder depuis Pharo.

4.7 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
```

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
```

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 réinitialiser le cache maintenu par Voyage. La réinitialisation se fait comme suit:

```
[ VORepository current reset
```

4.8 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.

Une interface Web en Seaside pour TinyBlog

Nous commençons par définir une interface telle que les utilisateurs la verrons. Dans un prochain chapitre nous développerons 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>.

Le travail présenté dans la suite est indépendant de celui sur Voyage et sur la base de données MongoDB.

5.1 Démarrer Seaside

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 ZnZincServerAdaptor, 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.

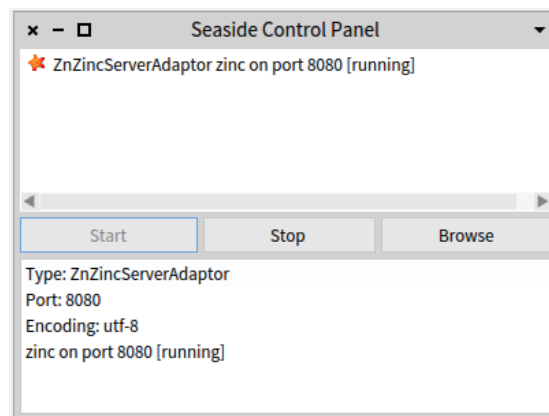


Figure 5-1 Lancer le serveur.

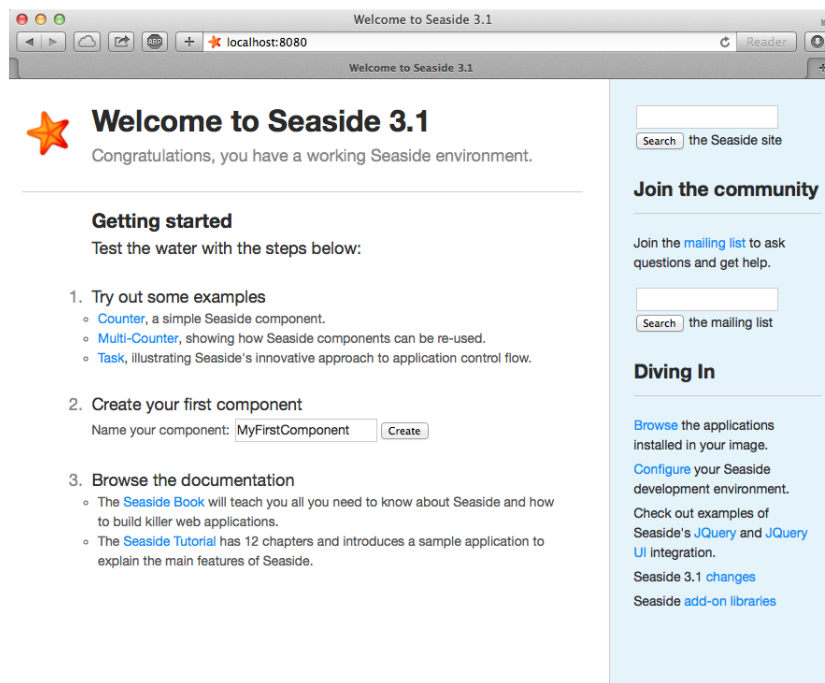


Figure 5-2 Vérification que Seaside fonctionne.

5.2 Point d'entrée de l'application

Créez la classe `TBApplicationRootComponent` qui est le point d'entrée de l'application. Elle sert à l'initialisation de l'application.

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

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

Les méthodes de classe `initialize` sont invoquées automatiquement lors du chargement de la classe.

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.

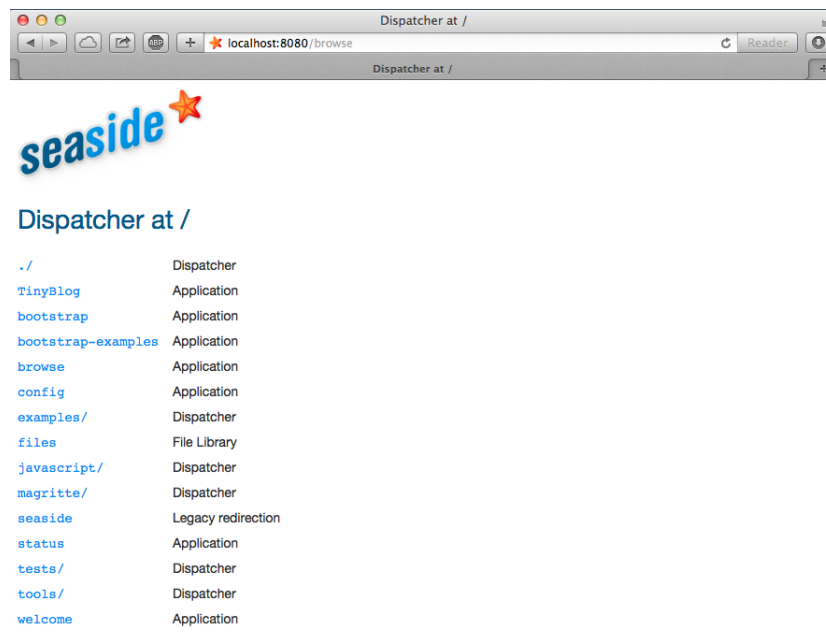


Figure 5-3 TinyBlog est bien enregistrée.

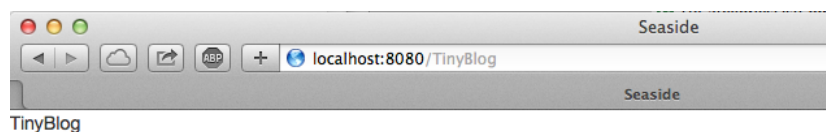


Figure 5-4 Une page quasi vide mais servie par Seaside.

```
[ TBAApplicationRootComponent 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 5-3.

5.3 Premier rendu simple

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

```
[ TBAApplicationRootComponent >> 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 5-4.

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

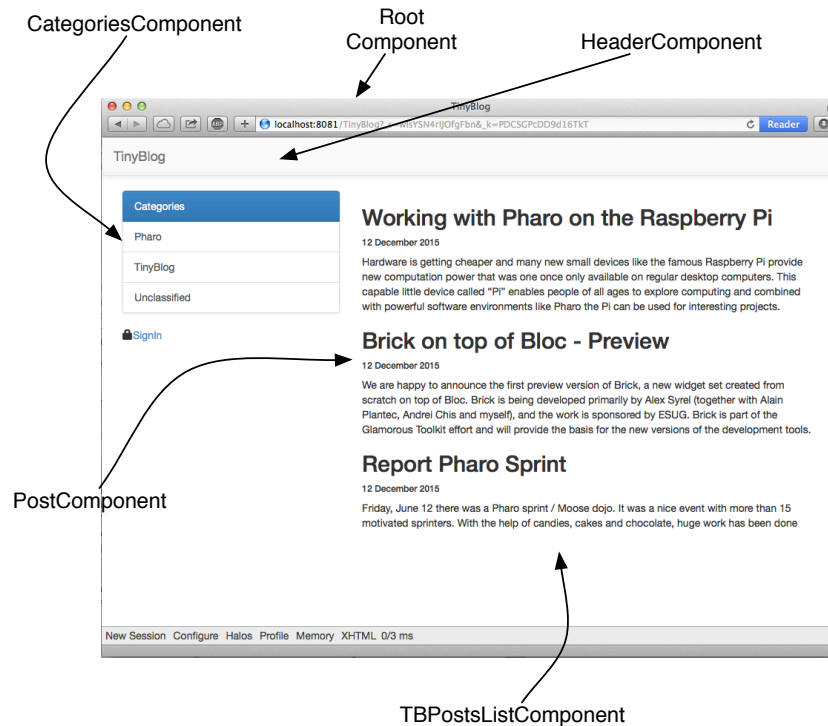


Figure 5-5 Les composants composant l'application TinyBlog.

```
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. 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.

Pour cela, nous avons décidé que le composant `TBApplicationRootComponent` contiendra des composants héritant tous de la classe abstraite `TBScreenComponent` que nous allons définir dans le prochain chapitre.

5.4 Composants visuels pour TinyBlog

Nous sommes maintenant prêts à définir les composants visuels de notre application Web. Les premiers chapitres de <http://book.seaside.st> peuvent vous y aider et compléter efficacement ce tutoriel.

La figure 5-5 montre les différents composants que nous allons développer 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.

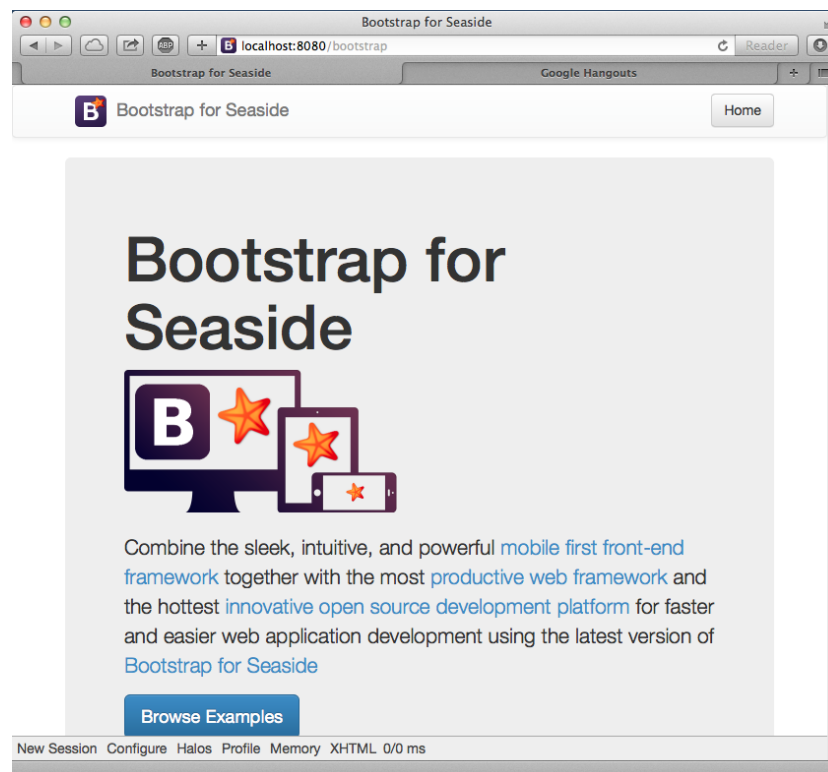


Figure 5-6 Accès à la bibliothèque Bootstrap.

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

Les différents composants d'interface de TinyBlog auront 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).

```
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
```

En inspectant l'objet `blog` retourné par `TBBlog current`, vérifier qu'il contient bien des posts. Si ce n'est pas le cas, exécuter `TBBlog createDemoPosts`.

Dans le futur, lorsque nous gérerons les utilisateurs et le fait qu'un utilisateurs puisse avoir plusieurs blogs nous modifierons cette méthode pour utiliser des informations stockées dans la session active (Voir `TBSession` plus loin).

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 class racine.
- En particulier nous exprimons la relation entre un composant et un sous composant en redéfinissant la méthode `children`.

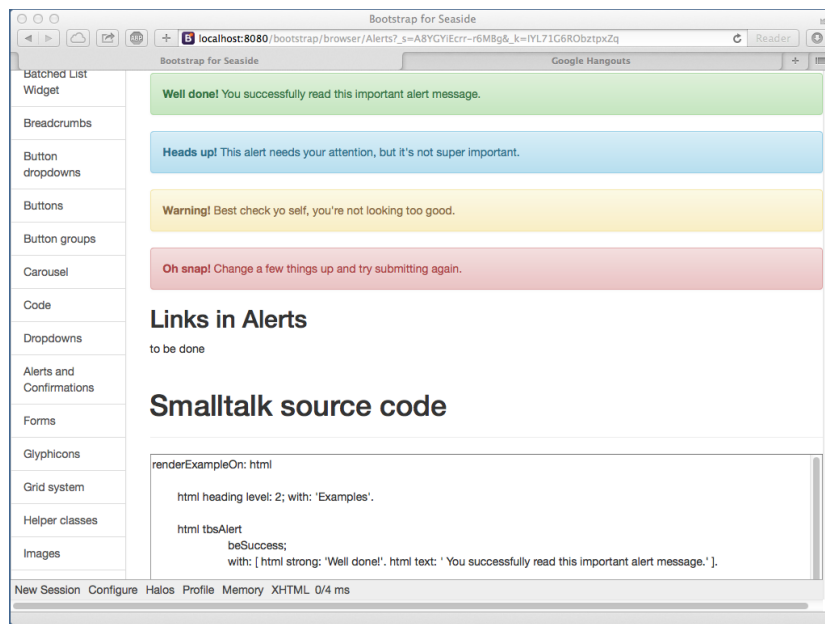


Figure 5-7 Comprendre un élément et son code en Bootstrap.

5.5 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 5-6.

Cliquer sur le lien **Exemples** au bas de la page et vous pouvez ainsi voir les éléments graphiques ainsi que le code pour les obtenir comme montré par la figure 5-7.

Bootstrap

Le repository pour le source et la documentation est <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.

5.6 Définition du composant TBHeaderComponent

Profitons de ce composant, pour insérer dans la partie supérieure de chaque composant l'instance d'un composant représentant l'entête de l'application. Nous appliquons le schéma présenté avant: définition d'une classe, référence depuis la class utilisatrice, redéfinition de la méthode `children`.

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

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

Le protocole `'rendering'` contient la méthode `renderContentOn:` chargée d'afficher l'entête.

```
TBHeaderComponent >> renderContentOn: html
  html tbsNavbar beDefault with: [
    html tbsNavbarBrand
```

```
[
    url: '#';
    with: 'TinyBlog' ]
```

L'entête (header) est affichée à l'aide d'une barre de navigation Bootstrap (voir la figure 5-8)

Par défaut dans une barre de navigation Bootstrap, il y a un lien sur `tbsNavbarBrand` qui est ici inutile (sur le titre de l'application). Ici nous l'initialisons avec une ancre '#' de façon à ce que si l'utilisateur clique sur le titre, il ne se passe rien. En général, cliquer sur le titre de l'application 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 le header pourrait afficher ce titre.

5.7 Utilisation du composant header

Il n'est pas souhaitable d'instancier systématiquement le composant à chaque fois qu'un composant est appelé. Créons une variable d'instance header dans `TBScreenComponent` que nous initialisons.

```
[WAComponent subclass: #TBScreenComponent
 instanceVariableNames: 'header'
 classVariableNames: ''
 package: 'TinyBlog-Components']
```

Créons une méthode `initialize` dans le protocole 'initialize-release':

```
[TBScreenComponent >> initialize
 super initialize.
 header := TBHeaderComponent new.]
```

5.8 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). Nous faisons cela en spécialisant la méthode `children`.

```
[TBScreenComponent >> children
 ^ OrderedCollection with: header]
```

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

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

5.9 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 `main` comme variable d'instance dans la classe `TBApplicationRootComponent`. Nous l'initialisons dans la méthode `initialize` suivante. Nous obtenons la situation décrite par la figure 5-9.

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

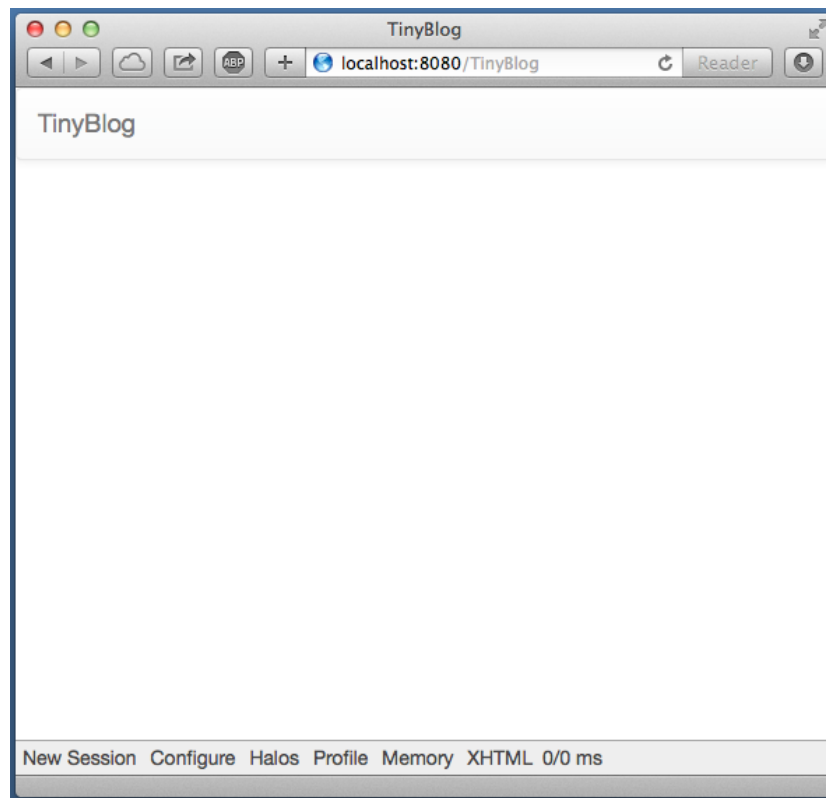


Figure 5-8 TinyBlog avec une barre de navigation.

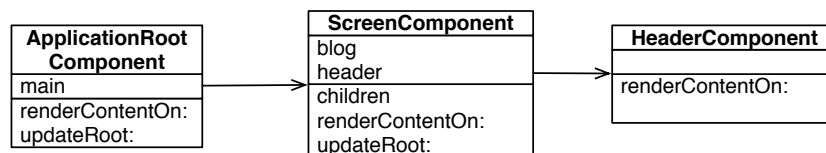


Figure 5-9 Le composant ApplicationRootComponent utilise de manière temporaire le composant ScreenComponent qui a un HeaderComponent.

```

[ TBAApplicationRootComponent >> renderContentOn: html
  html render: main

```

Nous déclarons aussi la relation de contenu en retournant main parmi les enfants de TBAApplicationRootComponent.

```

[ TBAApplicationRootComponent >> children
  ^ { main }

```

Si vous faites un rafraîchissement de l'application dans votre navigateur web vous devez voir la Figure 5-8.

Amélioration possibles

Le nom du blog doit être particularisable en utilisant par exemple une variable d'instance de la classe TBBlog et l'entête (header) pour afficher ce titre.

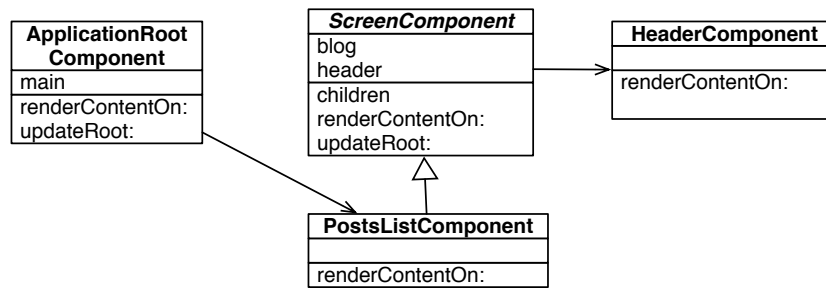


Figure 5-10 Le composant ApplicationRootComponent utilise le composant PostsListComponent.

5.10 Liste des posts

Nous allons afficher la liste des posts - ce qui reste d'ailleurs le but d'un blog. Ici nous parlons de l'accès public offert aux lecteurs du blog. Dans le futur, nous proposerons une interface d'administration des posts.

Créons un composant `TBPostsListComponent` qui hérite de `TBScreenComponent`:

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

Ajoutons une méthode `renderContentOn:` (protocole rendering) provisoire pour tester l'avancement de notre application (voir Figure 5-11).

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

Nous pouvons maintenant dire au composant de l'application d'utiliser ce composant comme illustré dans la figure 5-10. Pour cela nous modifions-la ainsi:

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

Editer cette méthode n'est pas une bonne pratique. Nous ajoutons une méthode `setter` qui nous permettra de changer dynamiquement de composant dans le futur tout en gardant le composant actuel pour une initialisation par défaut.

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

5.11 Le composant Post

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

Chaque post du blog sera représenté visuellement par une instance de `TBPostComponent` qui affiche le titre, la date et le contenu d'un post. Nous allons obtenir la situation décrite par la figure 5-12.

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

```
TBPostComponent >> initialize
  super initialize.
```

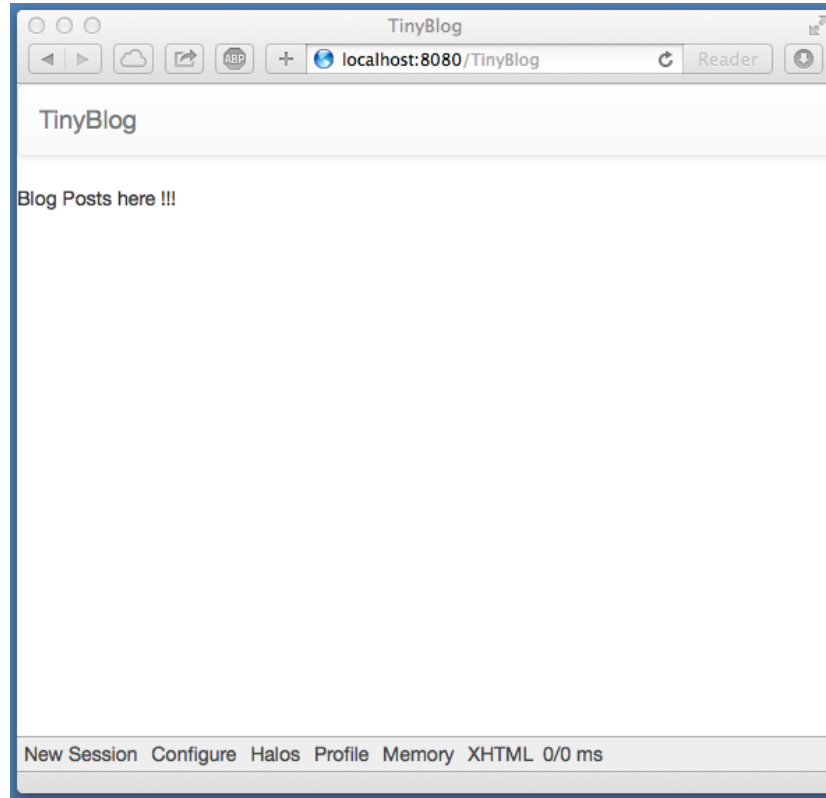


Figure 5-11 TinyBlog avec une liste de posts plutôt élémentaire.

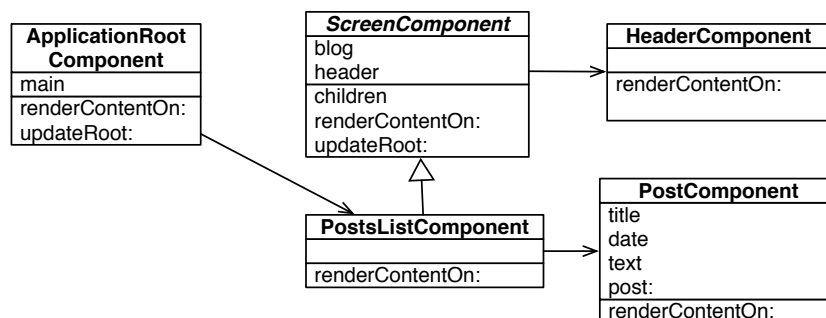


Figure 5-12 Ajout du composant Post.

```
[
    post := TBPPost new.
[TBPPostComponent >> title
    ^ post title
[TBPPostComponent >> text
    ^ post text
[TBPPostComponent >> date
    ^ post date
```

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

```
[TBPPostComponent >> 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 et qui utilise Magritte, nous montrerons qu’il est rare de définir un composant de manière aussi manuelle. En effet, Magritte permet de décrire les données manipulées et offre ensuite la possibilité de générer automatiquement des composants Seaside. Le code équivalent serait comme suit:

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

5.12 Afficher les posts

Maintenant nous pouvons afficher des posts présents dans la base.

Il ne reste plus qu’à modifier la méthode `TBPostsListComponent >> renderContentOn:` pour afficher l’ensemble des blogs 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 un message d’erreur.

5.13 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 5-13.

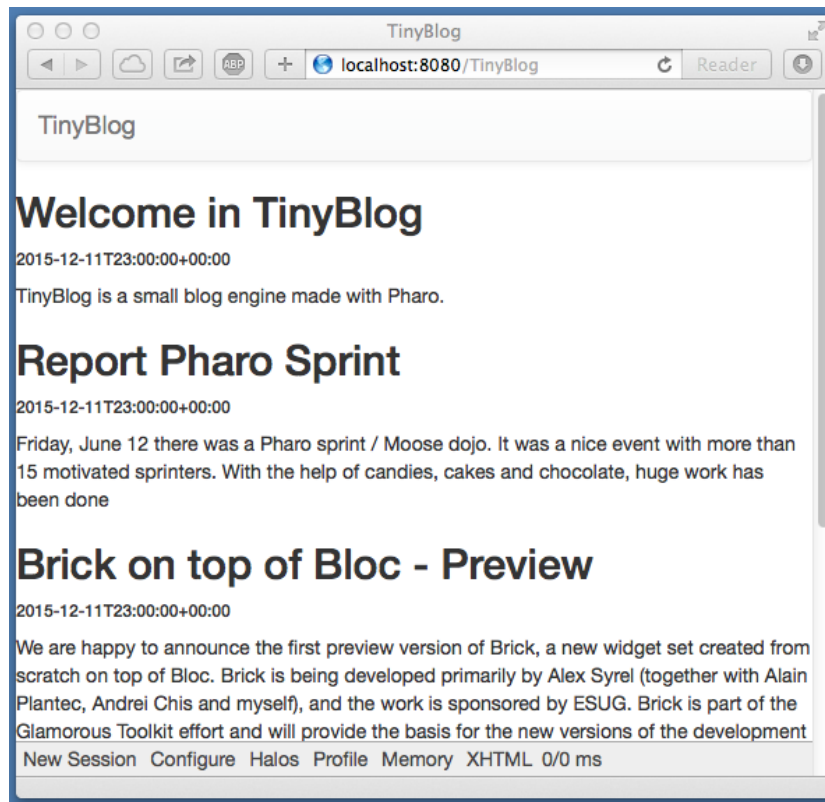


Figure 5-13 TinyBlog avec une liste de posts.

5.14 Affichage de la liste des posts avec Bootstrap

Nous allons utiliser Bootstrap pour rendre la liste un peu plus jolie en utilisant un container.

```
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 5-14.

5.15 Relation composite composants

Alors que la solution fonctionne, Seaside nécessite que les composants (ici TBPostComponent) d'un composite (ici TBPostsListComponent) soient accessibles via la méthode children. Nous allons donc transformer la solution précédente pour suivre le framework.

Nous ajoutons une variable d'instance postComponents.

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

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

La méthode postComponents calcule et rend les composants pour les bulletins.

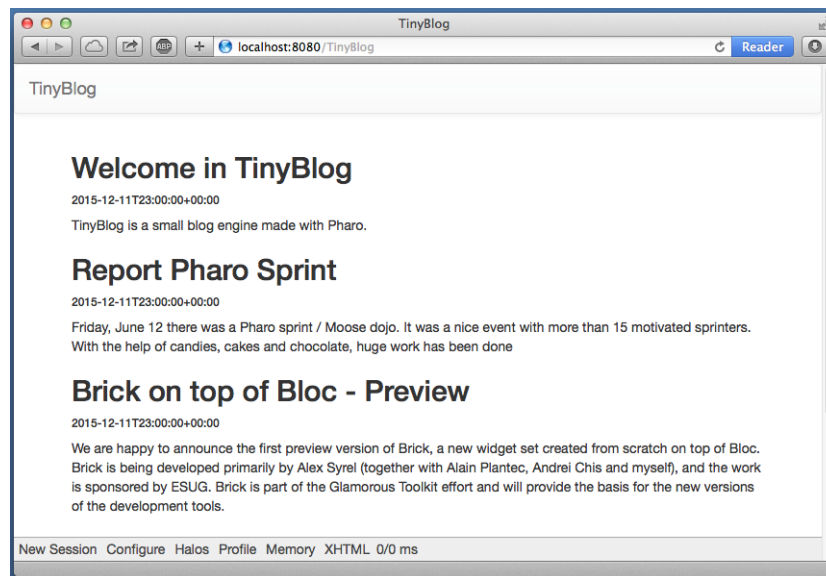


Figure 5-14 TinyBlog avec une liste de posts élémentaire.

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

TBPostsListComponent >> children
  ^ self postComponents
```

Et nous redéfinissons la méthode `renderContentOn:` pour utiliser les méthodes définies avant.

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

5.16 Affichage des posts par catégorie

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

Nous allons créer un composant pour gérer une liste de catégories nommée: `TBCategoriesComponent`.

Definition d'un composant pour les catégories

Nous avons besoin d'un composant qui affiche la liste des catégories présentes dans la base 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 5-15.

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

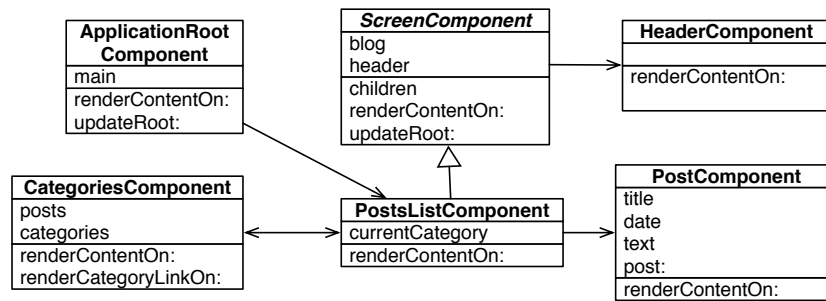


Figure 5-15 Ajout du composant Categories.

```

[ TBCategoriesComponent >> categories
  ^ categories

[ TBCategoriesComponent >> categories: aCollection
  categories := aCollection

[ TBCategoriesComponent >> postsList: aComponent
  postsList := aComponent

[ TBCategoriesComponent >> postsList
  ^ postsList
  
```

Nous définissons aussi une méthode de création au niveau classe.

```

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

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

```

[ TBScreenComponent subclass: #TBPPostsListComponent
  instanceVariableNames: 'currentCategory postComponents'
  classVariableNames: ''
  package: 'TinyBlog-Components'

[ TBPPostsListComponent >> currentCategory
  ^ currentCategory

[ TBPPostsListComponent >> currentCategory: anObject
  currentCategory := anObject
  
```

La méthode selectCategory:

La méthode selectCategory: (protocole 'action') communique au composant TBPPostsListComponent la nouvelle catégorie courante.

```

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

5.17 Rendu des catégories

Nous pouvons maintenant ajouter une méthode (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 d'un callback est un block et peut contenir n'importe quelle expression. Cela illustre la puissance de Seaside.

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

Reste maintenant à écrire la méthode de rendu du composant: On itère sur toutes les catégories et on les affiche

```
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 posts en fonction de la catégorie courante.

5.18 Mise à jour des Posts

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

La méthode `readSelectedPosts` récupère dans la base les bulletins à afficher. Si elle vaut nil, l'utilisateur n'a pas encore sélectionné une 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 la catégorie.

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

Nous redéfinissons la méthode `postComponents` pour utiliser la méthode `readSelectedPosts`.

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

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 postComponents do: [ :p |
            html render: p ] ]
```

Une instance du composant `TBCategoriesComponent` est ajoutée sur la page et permet de sélectionner la catégorie courante (voir la figure 5-16).

5.19 Look et agencement

Nous allons maintenant agencer le composant `TBPostsListComponent` en utilisant une mise en place d'un 'responsive design' pour la liste des posts. 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

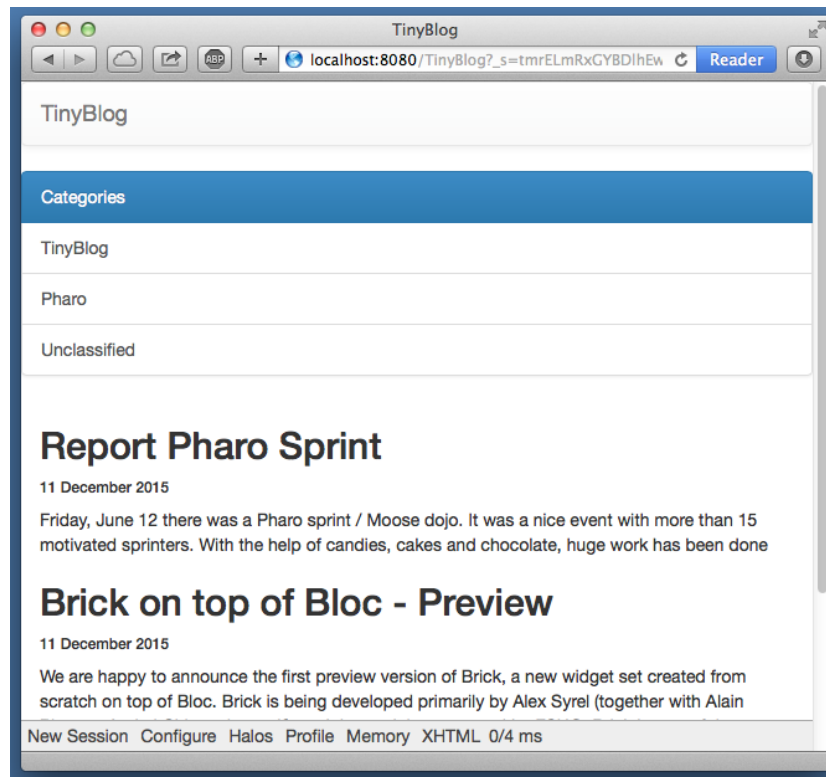


Figure 5-16 Catégories afin de sélectionner les posts.

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;
      mediumSize: 4;
      with: [ html
        render:
          (TBCategoriesComponent
            categories: self blog allCategories
            postsList: self) ].
      html tbsColumn
        extraSmallSize: 12;
        smallSize: 10;
        mediumSize: 8;
        with: [ self postComponents do: [ :p | html render: p ] ] ] ] ]
```

Vous devez obtenir une application proche de celle représentée figure 5-17.

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 :

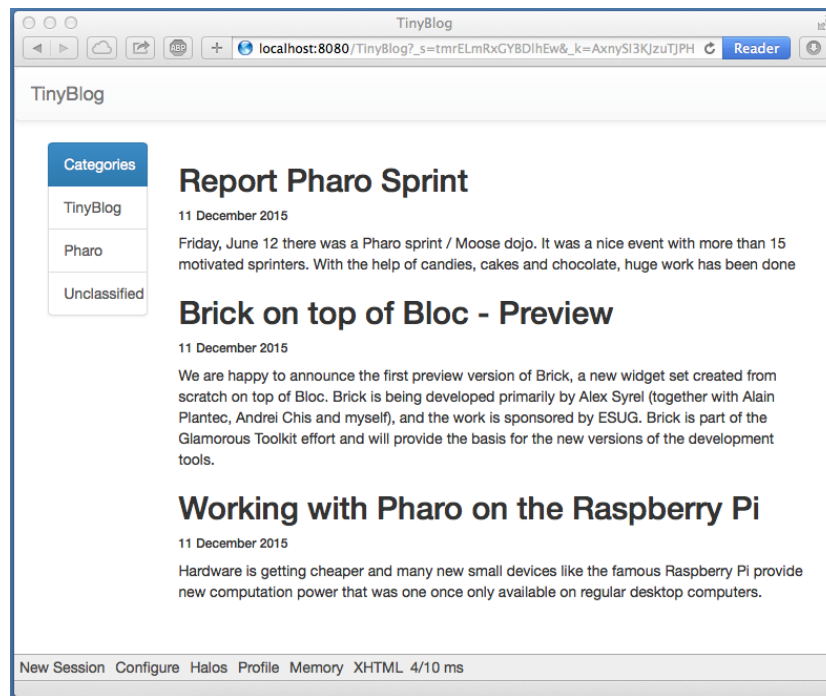


Figure 5-17 Avec un meilleur agencement.

```
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 `TBPostsListComponent >> renderContentOn: html` dans un tel état. Elle est bien trop longue et difficilement réutilisable. Proposer une solution.

5.20 Notre solution: Plein de petites méthodes

Nous allons découper la méthode en plusieurs petites méthodes.

```
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 >> basicRenderCategoriesOn: html
    html render: (TBCategoriesComponent
        categories: self blog allCategories
```

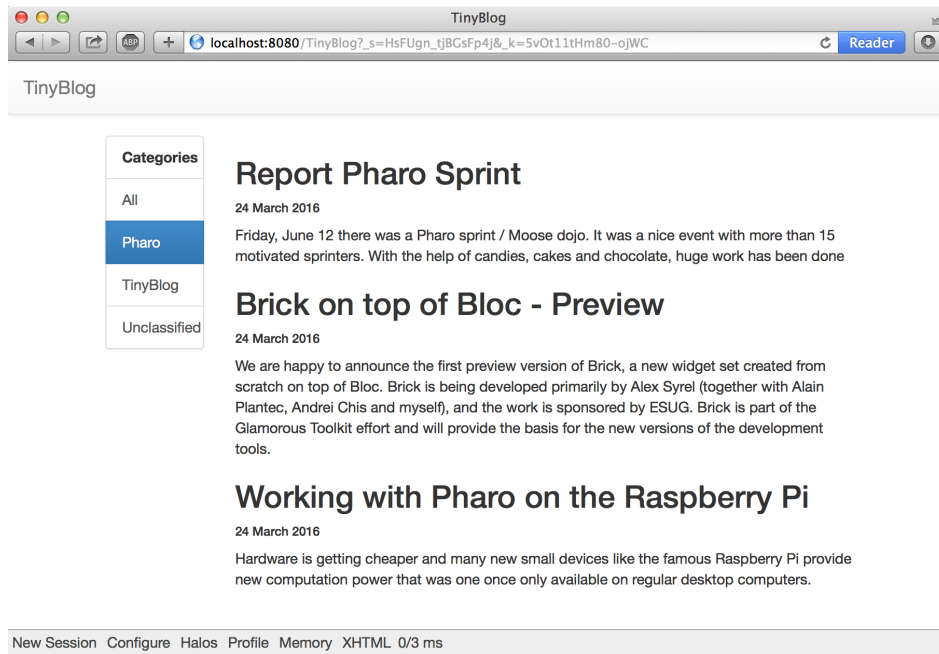


Figure 5-18 Final TinyBlog Public UI.

```

    postsList: self)
  TBPostsListComponent >> renderPostColumnOn: html
    html tbsColumn
      extraSmallSize: 12;
      smallSize: 10;
      mediumSize: 8;
      with: [ self basicRenderPostsOn: html ]
  TBPostsListComponent >> basicRenderPostsOn: html
    self postComponents do: [ :p | html render: p ]

```

Nous voici prêts à définir la partie administrative de l'application.

L'application finale devrait ressembler à la figure 5-18.

5.21 Conclusion

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`.

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 construisent de la même manière. Donc vous avez les bases pour définir de nombreuses applications web. Dans le chapitre suivant nous allons 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 :

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

Interface Web d'administration pour TinyBlog

Dans le chapitre précédent nous avons vu comment développer l'embryon d'une application : nous avons défini des composants qui interagissent entre eux. Chaque composant est alors responsable son état et de son rendu graphique. Alors que le chapitre précédent décrit l'essentiel du développement avec Seaside 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. Nous allons tout d'abord définir un composant d'identification : cette définition de composant va nous permettre d'illustrer comment la saisie de champs utilise de manière élégante les variables d'instances d'un composant ainsi que l'invocation de composants.

6.1 Administration de TinyBlog

Nous allons aborder maintenant l'administration de TinyBlog. Cet exercice va nous permettre de montrer comment utiliser des informations de session ainsi que Magritte pour la définition de rapports.

Le scénario assez classique que nous allons développer 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. Le lien permettant d'afficher le composant d'authentification sera placé sous la liste des catégories.

6.2 Composant d'identification

Nous allons commencer par développer un composant d'identification qui lorsqu'il sera invoqué ouvrira une boîte de dialogue pour demander les informations d'identification. Le résultat que nous voulons obtenir est montré dans la figure 6-1. Remarquer qu'une telle fonctionnalité devrait faire partie d'une bibliothèque de composants de base en Seaside. Le projet Heimdal disponible sur <http://www.github.com/DuneSt/> offre cette fonctionnalité.

Ce composant va nous permettre d'illustrer comment la saisie de champs utilise de manière élégante les variables d'instances du composant.

Définition du composant

Nous définissons une nouvelle classe sous-classes de la classe `WComponent` et des accesseurs. Ce composant contient un compte d'administration, le mot de passe associé ainsi que le composant

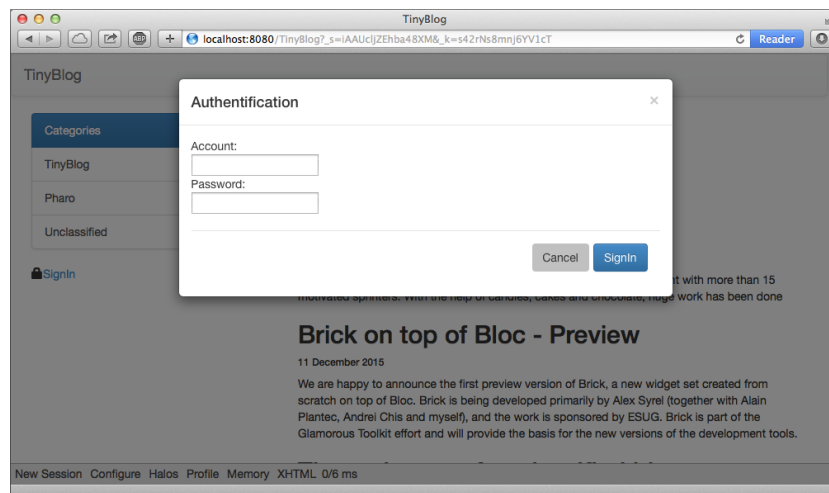


Figure 6-1 Authentification avec un meilleur agencement.

a invoqué pour accéder à l'administration.

```

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

TBAuthenticationComponent >> account
  ^ account

TBAuthenticationComponent >> account: anObject
  account := anObject

TBAuthenticationComponent >> password
  ^ password

TBAuthenticationComponent >> password: anObject
  password := anObject

TBAuthenticationComponent >> component
  ^ component

TBAuthenticationComponent >> component: anObject
  component := anObject

```

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

```

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

```

Rendu du composant.

La méthode `renderContentOn:` définit le contenu d'une boîte de dialogue modale. Il est composé d'un header et d'un corps.

```

TBAuthenticationComponent >> renderContentOn: html
  html tbsModal
    id: 'myAuthDialog';
    with: [
      html tbsModalDialog: [
        html tbsModalContent: [

```

```

[
    self renderHeaderOn: html.
    self renderBodyOn: html ] ] ]

```

Le header affiche un titre avec de larges fontes.

```

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

```

Le corps du composant affiche une 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 d'une instance du composant quand l'utilisateur confirme sa saisie. L'argument du callback: est un block avec un argument et c'est cet argument qui représente la valeur du text input.

```

[ TBAuthenticationComponent >> renderAccountFieldOn: html
  html
    tbsFormGroup: [ html label with: 'Account'.
      html textInput
        tbsFormControl;
        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 ]

```

Les boutons définissent par défaut le bouton nommé 'SignIn' et son action est de vérifier l'adéquation entre le mot de passe et le compte.

```

[ TBAuthenticationComponent >> renderButtonsOn: html
  html tbsSubmitButton value: 'Cancel'.
  html tbsSubmitButton
    bePrimary;
    callback: [ self validate ];
    value: 'SignIn'

```

Lorsque l'utilisateur clique sur le bouton 'SignIn', le message `validate` est envoyé: il vérifie que l'utilisateur a bien le compte 'admin' et a saisi le mot de passe 'password'.

```

[ TBAuthenticationComponent >> validate
  (self account = 'admin' and: [ self password = 'password' ])
    ifTrue: [ self gotoAdministration ]
[ TBAuthenticationComponent >> gotoAdministration

```

Améliorations.

Rechercher une autre méthode pour réaliser l'authentification de l'utilisateur (utilisation d'un backend de type base de données, LDAP ou fichier texte). En tout cas, ce n'est pas à la boîte de login de faire ce travail, il faut le déléguer à un objet métier qui saura consulter le backend et authentifier l'utilisateur.

De plus le composant `TBAuthenticationComponent` pourrait afficher l'utilisateur lorsque celui-ci est logué.

6.3 Intégration de l'authentification

Il faut maintenant intégrer le lien qui déclenchera l'affichage de la boîte modale d'authentification. Au tout début de la méthode `renderContentOn:` du composant `TBPostsListComponent`, on ajoute le rendu du composant d'authentification. Ce composant reçoit en paramètre la référence vers le composant affichant les posts. C'est depuis ce composant (`TBPostsListComponent`) que nous donnerons accès à l'administration.

```
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  html render: (TBAuthenticationComponent from: self).
  html
    tbsContainer: [
      html tbsRow
        showGrid;
        with: [ self renderCategoryColumnOn: html.
                self renderPostColumnOn: html ] ]
```

■ **To do** Why here we do not see the component. I do not get it

On définit maintenant une méthode qui affiche un pictogramme clé et un lien 'SignIn'.

```
TBPostsListComponent >> renderSignInOn: html
  html tbsGlyphIcon perform: #iconLock.
  html html: '<a data-toggle="modal" href="#myAuthDialog" class="link">SignIn</a>'
```

Nous ajoutons le composant d'authentification dessous la liste de categories.

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

Lorsque nous pressons sur le lien `SignIn` nous obtenons la figure 6-1.

6.4 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 sur <https://github.com/SquareBracketAssociates/Magritte>.

Dans ce chapitre, nous allons 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 `Sea-side`.

Descriptions

Les cinq méthodes suivantes sont dans le protocole '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 complétée.

```
TBPost >> descriptionText
  <magritteDescription>
  ^ MMemoDescription 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>
  ^ MDateDescription new
    accessor: #date;
    beRequired;
    yourself
```

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

```
TBPost >> descriptionVisible
  <magritteDescription>
  ^ MBooleanDescription 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.

6.5 Administration des posts

Nous allons développer deux composants. Le premier sera un rapport qui contiendra tous les posts et le second contiendra ce rapport. 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. Commençons donc par le composant d'administration.

Création d'un composant d'administration

Le composant TBAAdminComponent hérite de TBScreenComponent pour bénéficier du header et de l'accès au blog. Il contiendra en plus le rapport que nous construisons par la suite.

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

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

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

Nous modifions la méthode validate pour qu'elle invoque la méthode gotoAdministration définie dans le composant TBPostsListComponent. Cette dernière méthode invoque le composant d'administration.

```
TBPostsListComponent >> gotoAdministration
  self call: TBAAdminComponent new

TBAAuthenticationComponent >> validate
  (self account = 'admin' and: [ self password = 'password' ])
    ifTrue: [ self component gotoAdministration ]
```

Pour tester, identifiez-vous sur l'application et vous devez obtenir la situation telle que représentée par la figure 6-2.

6.6 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.

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

Avec la méthode from: nous disons que nous voulons créer un rapport en prenant les descriptions de n'importe quel blog.

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

On ajoute ensuite une variable d'instance report et ses accesseurs à la classe TBAAdminComponent.

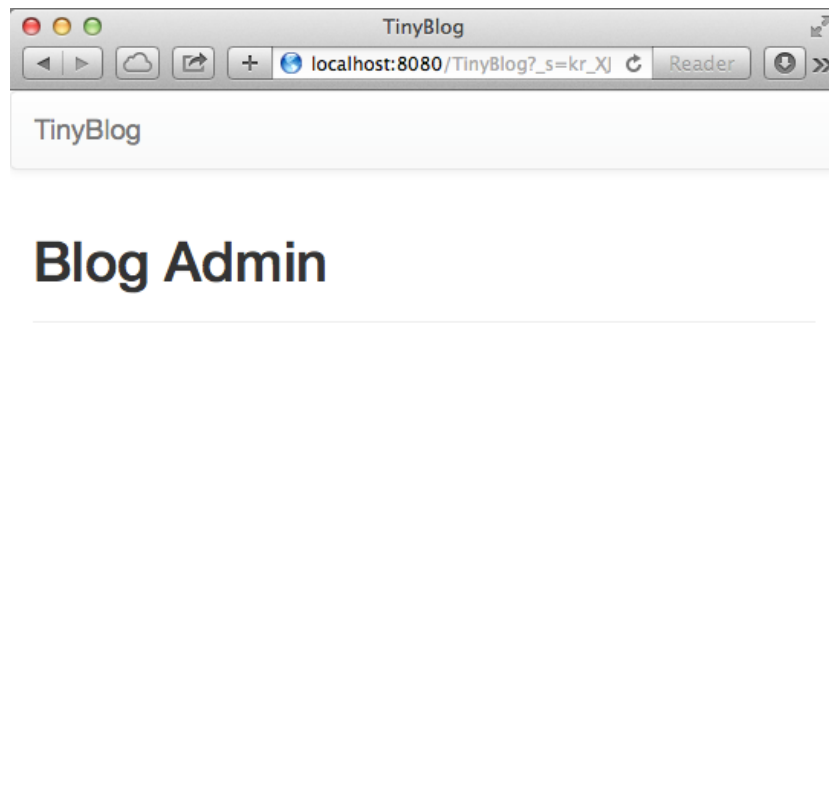


Figure 6-2 Un composant d'administration vide.

```
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.

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

La méthode `initialize` permet d'initialiser la définition du rapport. Nous fournissons au composant `TBPostReport` l'accès aux données.

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

Nous pouvons maintenant afficher le rapport.

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

Par défaut, le rapport affiche l'intégralité des données présentes dans chaque post mais 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
    ^ 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)
```

6.7 Amélioration des rapports

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 (il peut varier d'une instance à une autre). Pour gérer cela, il suffit de modifier les descriptions Magritte pour chaque variable d'instance.

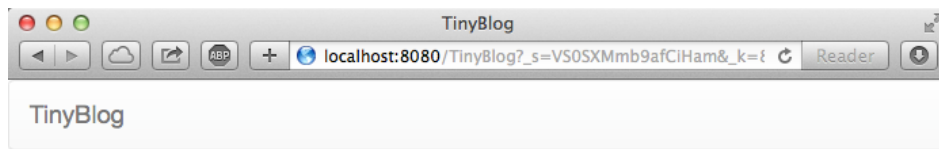
```
TBPost >> descriptionTitle
    <magritteDescription>
    ^ MStringDescription new
        label: 'Title';
        priority: 100;
        accessor: #title;
        beRequired;
        yourself
```

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

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

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

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

TinyBlog	Welcome in TinyBlog	2015-12-11T23:00:00+00:00
Pharo	Report Pharo Sprint	2015-12-11T23:00:00+00:00
Pharo	Brick on top of Bloc - Preview	2015-12-11T23:00:00+00:00
Unclassified	The sad story of unclassified blog posts	2015-12-11T23:00:00+00:00
Pharo	Working with Pharo on the Raspberry Pi	2015-12-11T23:00:00+00:00

Figure 6-3 Administration avec un rapport.

Identifiez-vous et vous devez obtenir la situation telle que représentée par la figure 6-3.

6.8 Gestion des posts

Nous pouvons maintenant mettre en place un CRUD (Create Read Update Delete) permettant de gérer les posts. Pour cela, nous allons ajouter une colonne (instance `MACCommandColumn`) au rapport qui regroupera les différentes opérations utilisant `addCommandOn:`.

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

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).
```

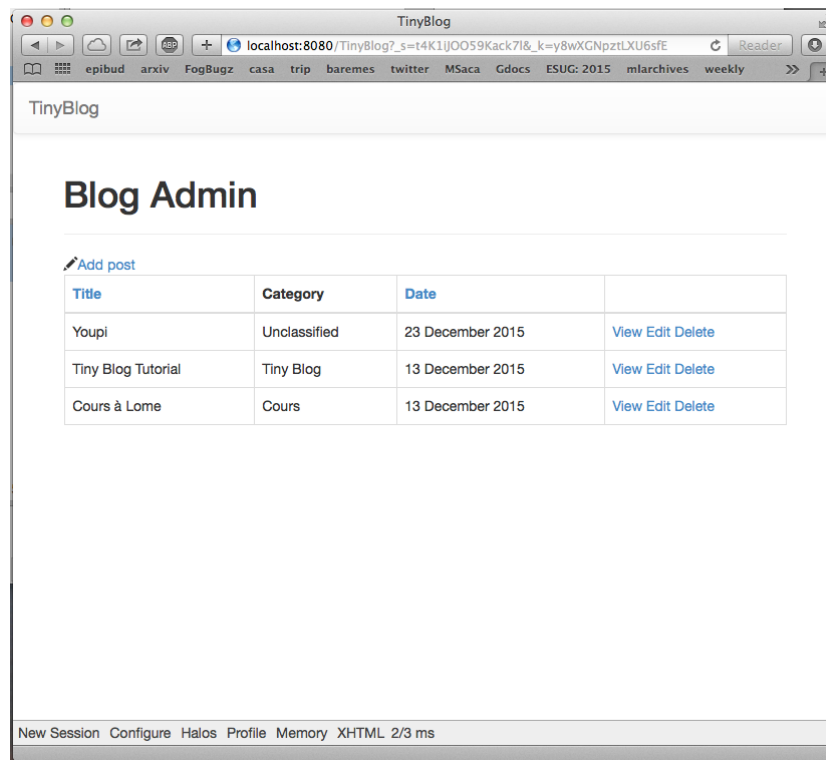


Figure 6-4 Ajout d'un post.

```
^ report
```

L'ajout (add) est dissocié des posts 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 perform: #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 6-4.

6.9 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).

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

La méthode `renderAddPostForm` illustre la puissance de Magritte pour générer des formulaires. Ici, le message `asComponent` envoyé à un objet métier instance de la classe `TBPost`, créé directe-

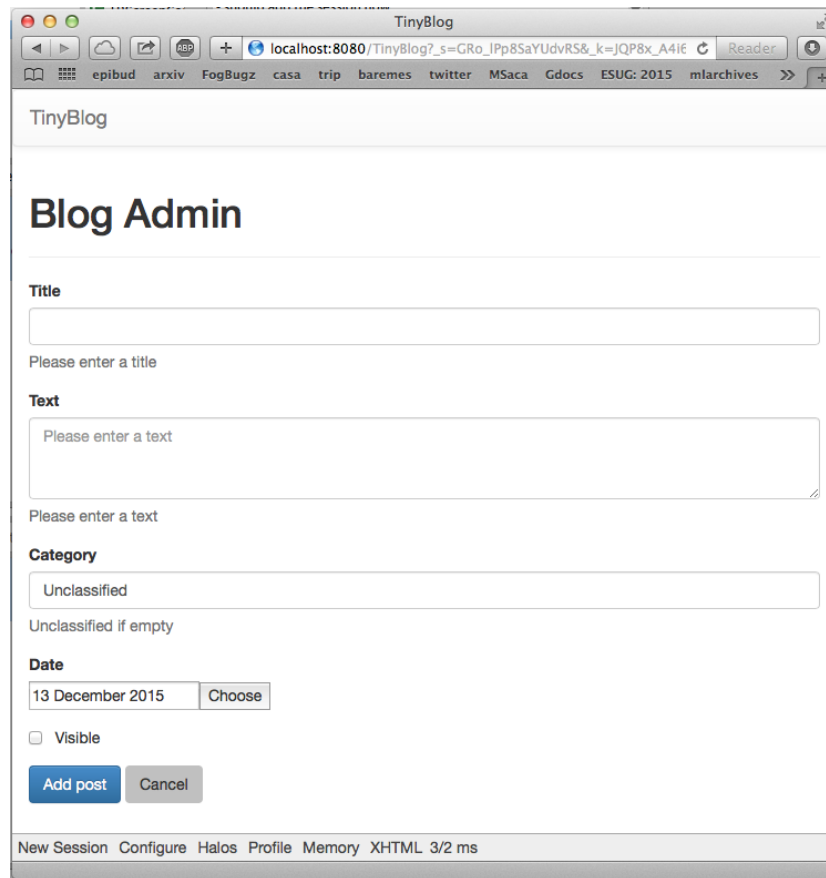


Figure 6-5 Ajout d'un post.

ment un composant Seaside. Nous ajoutons une décoration à ce composant Seaside afin de gérer ok/cancel.

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

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.

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 6-5.

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

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

TBPostsReport >> renderViewPostForm: aPost
  ^ aPost asComponent addDecoration: (
```

```

TBSMagritteFormDecoration buttons: { #cancel -> 'Back' });
yourself

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

```

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 ]

```

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

```

6.10 Gérer le problème du rafraîchissement des données

Les méthodes `TBPostsReport>>addPost:` et `TBPostsReport>>deletePost:` font bien leur travail mais les données à l'écran ne sont pas mises à jour. Il faut donc rafraîchir la liste des posts car il y a un décalage entre les données en mémoire et celles stockées dans la base de données.

```

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

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

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

```

Le formulaire est fonctionnel maintenant 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.

6.11 Amélioration de l'apparence du formulaire

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

```

TBPPost >> descriptionContainer
    <magritteContainer>

```

```

    ^ super descriptionContainer
      componentRenderer: TBSMagritteFormRenderer;
      yourself

```

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

```

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

TBPost >> descriptionText
  <magritteDescription>
  ^ 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

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

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

```

Le formulaire d'édition d'un post doit maintenant ressembler à celui de la figure 6-6.

6.12 Gestion de session

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. Une session est pratique pour gérer les informations de l'utilisateur en cours (identifié). Nous allons voir comment nous l'utilisons pour gérer une connexion.

L'administrateur du blog peut vouloir voyager entre la partie privée et la partie publique de Tiny-Blog.

Figure 6-6 Formulaire d'ajout d'un post avec Bootstrap.

Nous définissons une nouvelle sous-classe de `WASession` nommée `TBSession`. Pour savoir si l'utilisateur s'est authentifié, nous devons définir un objet session et ajouter une variable d'instance contenant une valeur booléenne précisant l'état de l'utilisateur.

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

```
TBSession >> logged
^ logged

TBSession >> logged: anObject
logged := anObject

TBSession >> isLogged
^ self logged
```

Il faut ensuite initialiser à 'false' cette variable d'instance à la création d'une session.

```
TBSession >> initialize
  super initialize.
  self logged: false.
```

Dans la partie privée de TinyBlog, ajoutons un lien permettant le retour à la partie publique. Nous utilisons ici le message `answer` puisque le composant d'administration a été appelé à l'aide du message `call`..

```
TBAdminComponent >> renderContentOn: html
  super renderContentOn: html.
  html tbsContainer: [
```

```

html heading: 'Blog Admin'.
html tbsGlyphIcon perform: #iconEyeOpen.
html anchor
    callback: [ self answer ];
    with: 'Public Area'.
html horizontalRule.
html render: self report.
]

```

Dans l'espace public, il nous faut modifier le comportement du lien permettant d'accéder à l'administration. Il doit provoquer l'affichage de la boîte d'authentification uniquement si l'utilisateur ne s'est pas encore connecté.

```

TBPostsListComponent >> renderSignInOn: html
    self session isLoggedIn
        iffFalse: [
            html tbsGlyphIcon perform: #iconLock.
            html html: '<a data-toggle="modal" href="#myAuthDialog"
class="link">SignIn</a>' ]
        ifTrue: [
            html tbsGlyphIcon perform: #iconUser.
            html anchor callback: [ self gotoAdministration ]; with: 'Private area' ]

```

Enfin, le composant TBAuthenticationComponent doit mettre à jour la variable d'instance logged de la session si l'utilisateur est bien un administrateur.

```

TBAuthenticationComponent >> validate
    (self account = 'admin' and: [ self password = 'password' ])
        ifTrue: [
            self session logged: true.
            component gotoAdministration ]

```

Il vous faut maintenant spécifier à Seaside qu'il doit utiliser l'objet TBSession comme objet de session courant pour l'application TinyBlog. Cette initialisation s'effectue dans la méthode initialize de la classe TBApplicationRootComponent que l'on modifie ainsi:

```

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

```

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

6.13 Améliorations possibles

- Ajouter un bouton "Déconnexion"
- gérer plusieurs comptes administrateur ce qui nécessite une amélioration des sessions qui devront conserver l'identification de l'utilisateur courant
- gérer plusieurs blogs

Déploiement de TinyBlog

7.1 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.

7.2 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 "Ephemeris 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

7.3 Préparation de l'image Pharo pour PharoCloud

Actuellement, PharoCloud ne supporte que les images Pharo 4.

Télécharger une image PharoWeb 4¹. Télécharger également la VM pour les images Pharo 4². Lancer cette image avec la VM et nous allons maintenant la configurer.

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

¹<https://ci.inria.fr/pharo-contribution/job/PharoWeb/PHARO=40,VERSION=stable,VM=vm/lastSuccessfulBuild/artifact/PharoWeb.zip>

²<http://get.pharo.org/vm40>

```
"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'.
```

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.
```

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'.

"Start HTTP server"
ZnZincServerAdaptor startOn: 8080.

"Register TinyBlog on Seaside"
TBApplicationRootComponent initialize.
```

Sauvegarder maintenant votre image Pharo (Menu World > save).

Puis, tester en local dans votre navigateur via l'URL <http://localhost:8080>.

7.4 Déploiement 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 7-1.

En cliquant sur l'URL publique fournie vous voir s'afficher votre application TinyBlog comme sur la figure 7-2.

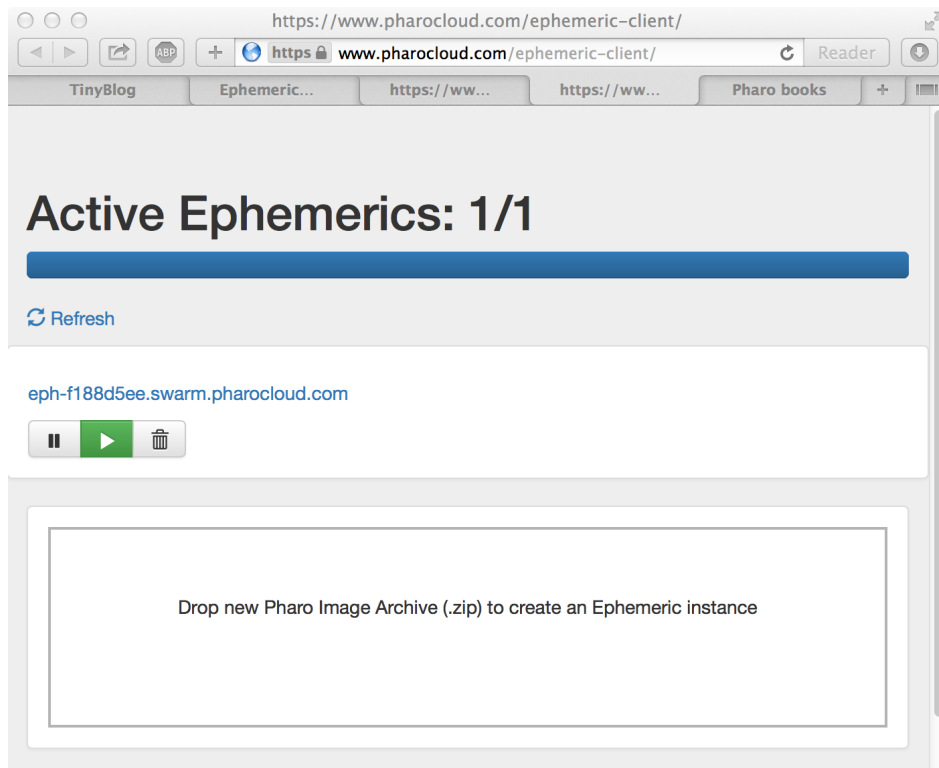


Figure 7-1 Administration des images Pharo sur Ephemeric Cloud.

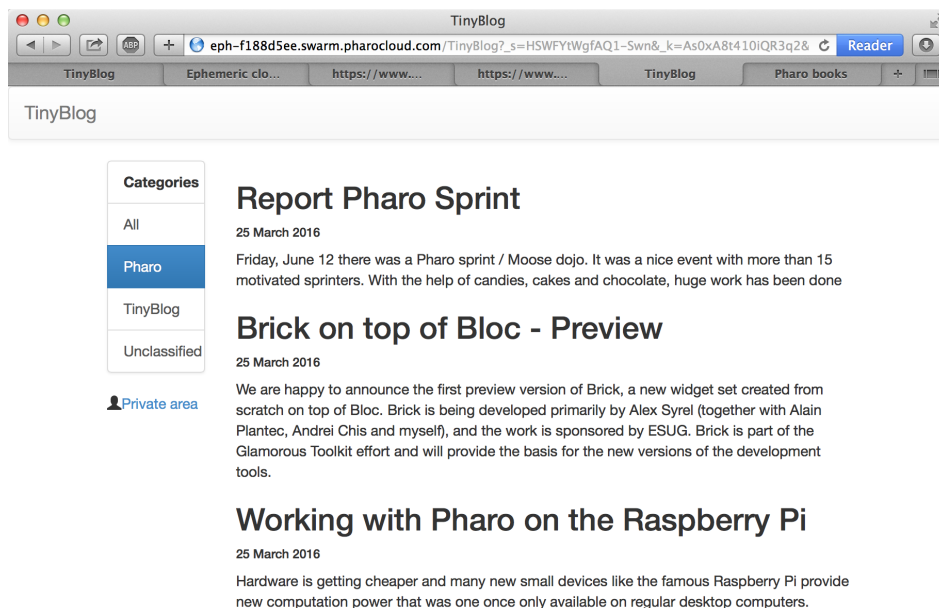


Figure 7-2 Votre application TinyBlog sur PharoCloud.

Part II

Éléments optionnels

Construire une interface Web avec Teapot pour TinyBlog

Ce chapitre optionnel peut être réalisé juste après le chapitre 3. Il vous guide dans la réalisation d'une interface web pour TinyBlog en utilisant Teapot (<http://smalltalkhub.com/#!/~zeroflag/Teapot>). Teapot est un serveur REST au dessus de Zinc - Un chapitre plus complet est disponible dans le livre *Entreprise Pharo: a Web perspective* (<http://books.pharo.org>). Il est plus primitif que Seaside qui permet de générer des éléments graphiques.

La classe TBTeapotWebApp

Créer une classe nommée TBTeapotWebApp ainsi:

```
Object subclass: #TBTeapotWebApp
  instanceVariableNames: 'teapot'
  classVariableNames: 'Server'
  package: 'TinyBlog-Teapot'
```

La variable teapot contiendra un petit server HTTP Teapot. Ici on utilise une implémentation différente du Design Pattern Singleton en utilisant une variable de classe nommée Server. Nous faisons cela afin de ne pas avoir deux serveurs gérant les connexions sur le même port.

Ajouter la méthode d'instance initialize pour initialiser la variable d'instance teapot :

```
TBTeapotWebApp >> initialize
  super initialize.
  teapot := Teapot configure: {
    #port -> 8081.
    #debugMode -> true }.
```

La Page d'accueil

Définissons une méthode homePage dans le protocol 'html' qui retourne le code HTML de la page d'accueil de notre application web. Commençons par une version simple :

```
TBTeapotWebApp >> homePage
  ^ '<html><body><h1>TinyBlog Web App</h1></body></html>'
```

Déclarer les URLs (routes)

Ajouter maintenant une méthode start pour que l'objet teapot réponde à des URLs particulières. Commençons par répondre à l'URL / lorsqu'elle est accédée en GET :

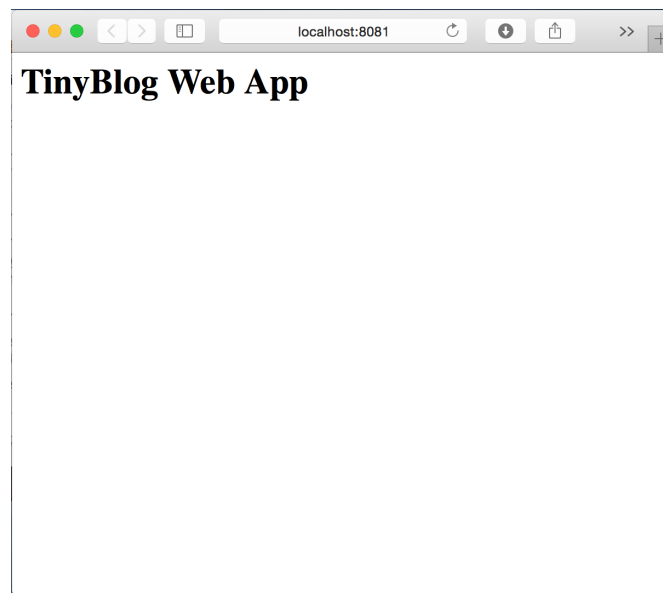


Figure 8-1 Une première page servie par notre application.

```
[ TBTeapotWebApp >> start
  "a get / is now returning an html welcome page"
  teapot
    GET: '/' -> [ self homePage ];
  start
```

Stopper l'application

Ajouter également une méthode pour stopper l'application :

```
[ TBTeapotWebApp >> stop
  teapot stop
```

Démarrage l'application

Ajouter deux méthodes start et stop côté classe pour respectivement démarrer et arrêter l'application dans le protocol 'start/stop'. Ces méthodes utilisent la variable de class Server pour implanter un Singleton.

```
[ TBTeapotWebApp class >> start
  Server ifNil: [ Server := self new start ]

[ TBTeapotWebApp class >> stop
  Server ifNotNil: [ Server stop. Server := nil ]
```

8.1 Tester votre application

Maintenant nous pouvons lancer notre application en exécutant le code suivant pour démarrer votre application :

```
[ TBTeapotWebApp start
```

Avec un navigateur web, vous pouvez accéder à l'application via l'URL <http://localhost:8081/>. Vous devriez voir s'afficher le texte "TinyBlog Web App" comme la figure 8-1.

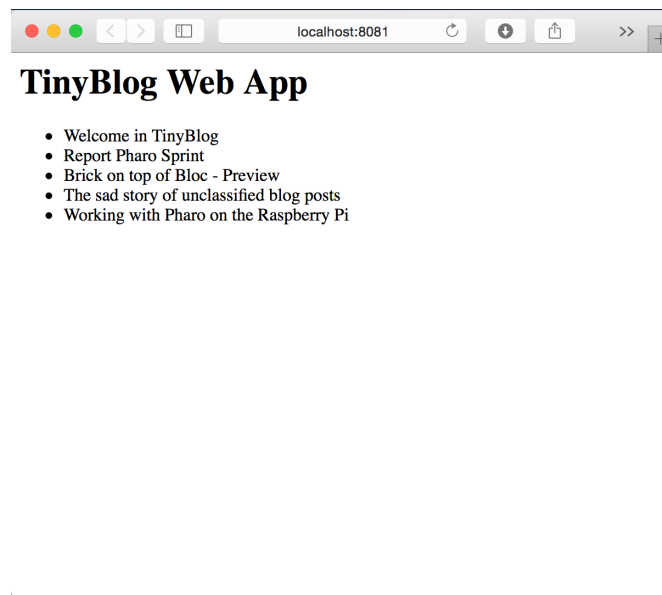


Figure 8-2 Afficher la liste des titres de posts.

8.2 Afficher la liste des posts

On souhaite maintenant modifier le code de la méthode `homePage` pour que la page d'accueil affiche la liste de tous les posts visibles. Pour rappel, tous les posts peuvent être obtenus via `TBBlog current allVisibleBlogPosts`. Ajoutons une méthode d'accès aux posts dans le protocole 'accessing' et modifions la méthode `homePage` ainsi que deux petites méthodes auxiliaires.

```
TBTeapotWebApp >> allPosts
  ^ TBBlog current allVisibleBlogPosts
```

Comme il faut générer une longue chaîne de caractères contenant le code HTML de cette page, nous utilisons un flût (stream) dans la méthode `homePage`. Nous avons également découpé le code en plusieurs méthodes dont `renderPageHeaderOn:` et `renderPageFooterOn:` qui permettent de générer l'en-tête et le pied de la page html.

```
TBTeapotWebApp >> homePage
  ^ String streamContents: [ :s |
    self renderPageHeaderOn: s.
    s << '<h1>TinyBlog Web App</h1>'.
    s << '<ul>'.
    self allPosts do: [ :aPost |
      s << ('<li>', aPost title, '</li>') ].
    s << '</ul>'.
    self renderPageFooterOn: s.
  ]
```

Notez que le message `<<` est un synonyme du message `nextPutAll:` qui ajoute une collection d'éléments dans un flût (stream).

```
TBTeapotWebApp >> renderPageHeaderOn: aStream
  aStream << '<html><body>'
```

```
TBTeapotWebApp >> renderPageFooterOn: aStream
  aStream << '</body></html>'
```

Tester l'application dans un navigateur web, vous devez maintenant voir la liste des titres des posts comme dans la figure 8-2. Si ce n'est pas le cas, assurez-vous que votre blog a bien des posts. Vous pouvez utiliser le message `createDemoPosts` pour ajouter quelques postes génériques.

```
[ TBBlog createDemoPosts
```

8.3 Détails d'un Post

Ajouter une nouvelle page

Améliorons notre application. On souhaite que l'URL `http://localhost:8081/post/1` permette de voir le post numéro 1.

Commençons par penser au pire, et définissons une méthode pour les erreurs. Nous définissons la méthode `errorPage` comme suit :

```
TBTeapotWebApp >> errorPage
  ^ String streamContents: [ :s |
    self renderPageHeaderOn: s.
    s << '<p>Oops, an error occurred</p>'.
    self renderPageFooterOn: s.
  ]
```

Teapot permet de définir des routes avec des patterns comme '`<id>`' dont la valeur est ensuite accessible dans l'objet requête reçu en paramètre du bloc.

Nous modifions donc la méthode `start` pour ajouter une nouvelle route à notre application permettant d'afficher le contenu d'un post.

```
TBTeapotWebApp >> start
  teapot
    GET: '/' -> [ self homePage ];
    GET: '/post/<id>' -> [ :request | self pageForPostNumber: (request at: #id)
      asNumber ];
  start
```

Il faut maintenant définir la méthode `pageForPostNumber` : qui affiche toutes les informations d'un post:

```
TBTeapotWebApp >> pageForPostNumber: aPostNumber
  |currentPost|
  currentPost := self allPosts at: aPostNumber ifAbsent: [ ^ self errorPage ].
  ^ String streamContents: [ :s |
    self renderPageHeaderOn: s.
    s << ('<h1>', currentPost title, '</h1>').
    s << ('<h3>', currentPost date mmdyyy, '</h3>').
    s << ('<p> Category: ', currentPost category, '</p>').
    s << ('<p>', currentPost text, '</p>').
    self renderPageFooterOn: s.
  ]
```

Vous devez maintenant redémarrer le serveur avant de tester votre application en accédant à l'URL: `http://localhost:8081/post/1`.

Dans le code ci-dessus, on peut voir que le nombre passé dans l'URL est utilisé comme la position du post à afficher dans la collection des posts. Cette solution est simple mais fragile puisque si l'ordre des posts dans la collection change, une même URL ne désignera plus le même post.

Ajouter des liens vers les posts

Modifions la méthode `homePage` pour que les titres des posts soient des liens vers leur page respective.

```
TBTeapotWebApp >> homePage
  ^ String streamContents: [ :s |
    self renderPageHeaderOn: s.
```

```

s << '<h1>TinyBlog Web App</h1>'.
s << '<ul>'.
self allPosts withIndexDo: [ :aPost :index |
    s << '<li>';
    << ('<a href="/post/', index asString, '>');
    << aPost title ;
    << '</a></li>' ].
s << '</ul>'.
self renderPageFooterOn: s
]

```

Maintenant, la page d'accueil de l'application affiche bien une liste de lien vers les posts.

8.4 Amélioration possibles

Cette application est un exemple pédagogique à travers lequel vous avez manipulé des collections, des flôts (Streams), etc.

Plusieurs évolutions peuvent être apportées telles que:

- sur la page de détails d'un post, ajouter un lien pour revenir à la page d'accueil,
- ajouter une page affichant la liste cliquable des catégories de posts,
- ajouter une page affichant tous les posts d'une catégorie donnée,
- ajouter des styles CSS pour avoir un rendu plus agréable.

Une interface REST pour TinyBlog

Ce chapitre décrit comment doter notre application TinyBlog d'une interface REST (REpresentational State Transfer). Le code est placé dans un package 'TinyBlog-Rest' car l'utilisation de REST est optionnelle. Les tests seront dans le package 'TinyBlog-Rest-Tests'.

9.1 Notions de base sur REST

REST se base sur les verbes HTTP pour décrire l'accès aux ressources HTTP (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>). Les principaux verbes ont la signification suivante:

* GET pour lire une ressource, * POST pour créer une nouvelle ressource, * PUT pour modifier une ressource existante, * DELETE pour effacer une ressource,

Les ressources sont définies à l'aide des URL qui pointent sur une entité. Le chemin précisé dans l'URL permet de donner une signification plus précise à l'action devant être réalisée. Par exemple, un GET /files/file.txt signifie que le client veut accéder au contenu de l'entité nommée file.txt. Par contre, un GET /files/ précise que le client veut obtenir la liste des entités contenues dans l'entité files.

Une autre notion importante est le respect des formats de données acceptés par le client et par le serveur. Lorsqu'un client REST émet une requête vers un serveur REST, il précise dans l'entête de la requête HTTP la liste des types de données qu'il est capable de gérer. Le serveur REST se doit de répondre dans un format compréhensible par le client et si cela n'est pas possible, de préciser au client qu'il n'est pas capable de lui répondre.

La réussite ou l'échec d'une opération est basée sur les codes de statut du protocole HTTP (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>). Par exemple, si une opération réussit, le serveur doit répondre un code 200 (OK). De même, si une ressource demandée par le client n'existe pas, il doit retourner un code 404 (Not Found). Il est très important de respecter la signification de ces codes de statut afin de mettre en place un dialogue compréhensible et normalisé entre le client et le serveur.

9.2 Définir un filtre REST

Pour regrouper les différents services REST de TinyBlog, il est préférable de créer un paquet dédié, nommé TinyBlog-REST. L'installation de ces services REST sera ainsi optionnelle. Si le paquet TinyBlog-REST est présent, le serveur TinyBlog autorisera: * l'obtention des l'ensemble des posts existants, * l'ajout d'un nouveau post, * la recherche parmi les posts en fonction du titre, * la recherche parmi les posts en fonction d'une période.

L'élément central de REST est un objet destiné à filtrer les requêtes HTTP reçues par le serveur et à déclencher les différents traitements. Si l'on C'est un quelque sorte une gare de triage permettant

d'aiguiller la requête du client vers le code apte à le gérer. Cet objet, nommé `TBRestfulFilter`, hérite de la classe `WRestfulFilter`.

```
WRestfulFilter subclass: #TBRestfulFilter
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'TinyBlog-REST'
```

Pour l'utiliser, il nous faut le déclarer au sein de l'application TinyBlog. Pour cela, éditez la méthode de classe `initialize` de la classe `TBApplicationRootComponent` pour ajouter une instance de `TBRestfulFilter`.

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

  app addFilter: TBRestfulFilter new.
```

N'oubliez pas d'initialiser à nouveau la classe `TBApplicationRootComponent` en exécutant la méthode `initialize` dans le Playground. Sans cela, Seaside ne prendra pas en compte le filtre ajouté.

```
TBApplicationRootComponent initialize
```

A partir de maintenant, nous pouvons commencer à implémenter les différents services REST.

9.3 Obtenir la liste des posts

Le premier service proposé sera destiné à récupérer la liste des posts. Il s'agit d'une opération de lecture et elle utilisera donc le verbe GET du protocole HTTP. La réponse sera produite au format JSON. La méthode `listAll` est marquée comme étant un point d'entrée REST à l'aide des annotations `<get>` et `<produces:>`.

Si le client interroge le serveur à l'aide de l'URL `http://localhost:8080/TinyBlog/listAll`, la méthode `listAll` est appelée. Celle-ci retourne les données selon le type MIME (Multipurpose Internet Mail Extensions) spécifié par l'annotation `<produces:>`.

PENSER A CHARGER SEASIDE-JSON-CORE ET SEASIDE-PHARO-JSON-CORE

```
TBRestfulFilter >> listAll
  <get>
  <produces: 'application/json'>
```

Afin de faciliter l'utilisation d'un service REST, il est préférable de préciser finement la ou les ressources manipulées. Dans le cas présent, le nom de la méthode `listAll` ne précise pas au client quelles sont les ressources qui seront retournées. Certes, nous savons que ce sont les posts mais après tout, cela pourrait également être des rubriques. Il faut donc être plus explicite dans la formalisation de l'URL afin de lui donner une réelle signification sémantique. C'est d'ailleurs la principale difficulté dans la mise en place des services REST. La meilleure méthode est de faire simple et de s'efforcer d'être cohérent dans la désignation des chemins d'accès aux ressources. Si nous voulons la liste des posts, il nous suffit de demander la liste des posts. L'URL doit donc avoir la forme suivante:

```
http://localhost:8080/TinyBlog/Posts
```

Pour obtenir cela, nous pouvons renommer la méthode `listAll` ou préciser le chemin d'accès qui appellera cette méthode. Cette seconde approche est plus souple puisqu'elle permet de réorganiser les appels aux services REST sans nécessiter de refactoriser le code.

```
TBRestfulFilter >> listAll
  <get>
  <path: '/posts'>
  <produces: 'application/json'>
```

Maintenant que nous avons défini le point d'entrée, nous pouvons implémenter la partie métier du service `listAll`. C'est à dire le code chargé de construire la liste des posts contenus dans la base. Une représentation astucieuse d'un service peut être réalisée à l'aide des objets. Chaque service REST sera contenu dans un objet distinct. Ceci facilitera grandement la maintenance et la compréhension du code.

La méthode `listAll` ci-dessous fait maintenant appel au service adéquat, nommé `TBRestServiceListAll`. Il est nécessaire de transmettre le contexte d'exécution de `Seaside` à l'instance de cet objet. Ce contexte est l'ensemble des informations transmises par le client REST (variables d'environnement HTTP ainsi que les flux d'entrée/sortie de `Seaside`).

```
TBRestfulFilter >> listAll
  <get>
  <path: '/posts'>
  <produces: 'application/json'>

  TBRestServiceListAll new applyServiceWithContext: (self requestContext)
```

9.4 Créer des Services

Ce contexte d'exécution sera utile pour l'ensemble de services REST de `TinyBlog`. Cela signifie donc que nous devons trouver une solution pour éviter la copie de sections de code identiques au sein des différents services. Pour cela, la solution évidente en programmation objet consiste à mettre en oeuvre un mécanisme d'héritage. Chaque service REST héritera d'un service commun nommé ici `TBRestService`. Ce service dispose de deux variables d'instance. `context` contiendra le contexte d'exécution et `result` recevra les éléments de réponse devant être transmis au client.

```
Object subclass: #TBRestService
  instanceVariableNames: 'result context'
  classVariableNames: ''
  category: 'TinyBlog-REST'

TBRestService >> context
^ context

TBRestService >> context: anObject
context := anObject
```

La méthode `initialize` assigne un conteneur de réponses à la variable d'instance `result`. Ce conteneur est l'objet `TBRestResponse`. Nous décrirons son implémentation un peu plus tard.

```
TBRestService >> initialize
  super initialize.
  result := TBRestResponseContent new.
```

Le contexte d'exécution est transmis au service REST à l'aide de la méthode `applyServiceWithContext:`. Une fois reçu, le traitement spécifique au service est déclenché à l'aide de la méthode `execute`. Au sein de l'objet `TBRestService`, la méthode `execute` doit être déclarée comme abstraite puisqu'elle n'a aucun travail à faire. Cette méthode devra être implémentée de manière spécifique dans les différents services REST de `TinyBlog`.

```

TBRestService >> applyServiceWithContext: aRequestContext
    self context: aRequestContext.
    self execute.

TBRestService >> execute
    self subclassResponsibility

```

Tous les services REST de TinyBlog doivent être capable de retourner une réponse au client et de lui préciser le format des données utilisé. Vous devez donc ajouter une méthode pour faire cela. Il s'agit de la méthode `dataType:with:`. Le premier paramètre sera le type MIME utilisé et le second, contiendra les données transmises au client. La méthode insère ces informations dans le flux de réponse fournit par Seaside. La méthode `greaseString` appliquée sur le type de données permet d'obtenir une représentation du type MIME sous la forme d'une chaîne de caractères (par exemple: "application/json").

```

TBRestService >> dataType: aDataType with: aResultSet
    self context response contentType: (aDataType greaseString).
    self context respond: [ :response | response nextPutAll: aResultSet ]

```

Avant de terminer l'implémentation de `TBRestServiceListAll`, il nous faut définir l'objet contenant les données devant être transmises au client. Il s'agit de `TBRestResponseContent`.

9.5 Construire une réponse

Un service REST doit pouvoir fournir sa réponse au client selon différents formats en fonction de la capacité du client à les comprendre. Un bon service REST doit être capable de s'adapter pour être compris par le client qui l'interroge. C'est pourquoi, il est courant qu'un même service puisse répondre dans les formats les plus courants tels que JSON, XML ou encore CSV. Cette contrainte doit être gérée dans notre application par l'utilisation d'un objet destiné à contenir les données. Au terme de l'exécution du service REST, c'est son contenu qui sera transformé dans le format adapté pour être ensuite transmis au client.

Dans TinyBlog, c'est l'objet `TBRestResponseContent` qui a la responsabilité de contenir les données à l'aide de la méthode d'instance `data`.

```

Object subclass: #TBRestResponseContent
    instanceVariableNames: 'data'
    classVariableNames: ''
    category: 'TinyBlog-REST'

```

Les données sont stockées au sein d'une collection ordonnée, initialisée à l'instanciation de l'objet. La méthode `add:` permet d'ajouter un nouvel élément à cette collection.

```

TBRestResponseContent >> initialize
    super initialize.
    data := OrderedCollection new.

TBRestResponseContent >> add: aValue
    data add: aValue

```

Nous avons également besoin de traducteurs pour convertir les données de la collection vers le format attendu par le client. Pour le format JSON, c'est la méthode `toJson` qui effectue le travail.

```

TBRestResponseContent >> toJson
    ^String streamContents: [ :stream |
        (NeoJSONWriter on: stream)
        for: Date
        customDo: [ :mapping | mapping encoder: [ :value | value asDateAndTime printString ] ];
        nextPut: data ]

```


Pourquoi ne pas ajouter d'autres traducteurs ? Pharo supporte parfaitement XML ou encore CSV. Nous vous laissons le soin d'ajouter ces formats aux services REST de TinyBlog.

9.6 Implémenter le code métier du service listAll

A ce stade, nous avons mis en place toute l'infrastructure qui permettra le bon fonctionnement des différents services REST de TinyBlog. L'implémentation de `listAll` va maintenant être rapide et extrêmement simple. En fait, nous n'avons besoin qu'une seule et unique méthode. Souvenez vous, c'est la méthode `execute` qui doit être ici implémentée.

```
TBRestService >> execute
  TBBlog current allBlogPosts do: [ :each | result add: (each asDictionary) ].
  self dataType: (WAMimeType applicationJson) with: (result toJson)
```

Cette méthode va collecter les posts présents dans la base de données de TinyBlog et les ajouter à l'instance de `TBRestResponseContent`. Une fois l'opération terminée, la réponse est convertie au format JSON puis retournée au client.

9.7 Utiliser un service REST

Je suis certain que vous vous posez la question de savoir comment utiliser ce service REST ! En fait, il y a différentes méthodes pour le faire.

En ligne de commande

Tout d'abord, si vous êtes un adepte du shell et des commandes Unix, il vous suffit d'utiliser les commandes `wget` ou `curl`. Celles-ci permettent d'envoyer une requête HTTP à un serveur.

Par exemple, la commande `wget` suivante interroge une instance locale de TinyBlog.

```
[wget http://localhost:8080/TinyBlog/posts
```

Les posts sont enregistrés dans un fichier nommé `posts` qui contient les données au format JSON.

```
[{"title":"A title","date":"2017-02-02T00:00:00+01:00","text":"A
  text","category":"Test"}, {"title":"un test de
  TinyBlog","date":"2017-02-03T00:00:00+01:00","text":"Incroyable, il n'a jamais été
  plus facile de faire un blog !","category":"Vos avis"}]
```

Avec un client graphique

Une autre approche, plus confortable et adaptée à la mise au point de vos services REST, consiste à utiliser un client graphique. Il en existe un grand nombre sur tous systèmes d'exploitation. Certains proposent des fonctionnalités avancées telles qu'un éditeur de requêtes HTTP ou HTTPS, la gestion de bibliothèques de requêtes ou encore la mise en place de tests unitaires. Nous vous recommandons de vous intéresser plus particulièrement à des produits fonctionnant directement avec des technologies web, sous la forme d'applications ou d'extensions intégrées à votre navigateur web.

Avec Zinc

Bien évidemment, il vous est possible d'interroger vos services REST directement avec Pharo. Le framework Zinc permet de le faire en une seule ligne de code.

```
[(ZnEasy get: 'http://localhost:8080/TinyBlog/posts') contents
```

Il vous est donc aisé de construire des services REST et d'écrire en Pharo des applications qui les consomment.

9.8 Recherche d'un Post

Maintenant nous allons proposer d'autres fonctionnalités comme la recherche d'un post. Nous définissons donc cette fonctionnalité dans la classe `TBlog`. La méthode `postWithTitle:` reçoit une chaîne de caractères comme unique argument et recherche un post ayant un titre identique à la chaîne de caractères. Si plusieurs posts sont trouvés, la méthode retourne le premier sélectionné.

```
TBlog >> postWithTitle: aString
| result |
result := self allVisibleBlogPosts select: [ :post | post title = aTitle ].
result ifNotEmpty: [ ^result first ] ifEmpty: [ ^nil ]
```

Il faut déclarer la route HTTP permettant de lancer la recherche. L'emplacement du titre recherché au sein de l'URL est entouré à l'aide d'accolades et le nom de l'argument doit être identique à celui du paramètre reçu par la méthode.

```
search: aTitle
<get>
<path: '/posts/search?title={aTitle}'>
<produces: 'application/json'>
```

La partie métier du service est implémentée dans l'objet `TBRestServiceSearch` qui hérite de `TBRestService`. Cet objet a besoin de connaître le titre du post recherché et fait appel à la méthode `TBlog » postWithTitle:` définie précédemment.

```
TBRestService subclass: #TBRestServiceSearch
instanceVariableNames: 'title'
classVariableNames: ''
category: 'TinyBlog-Rest'

TBRestServiceSearch >> title
^ title

TBRestServiceSearch >> title: anObject
title := anObject

TBRestServiceSearch >> execute
| post |

post := TBlog current postWithTitle: title urlDecoded.

post
ifNotNil: [ result add: (post asDictionary) ]
ifNil: [ self context response notFound ].
self dataType: (WAMimeType applicationJson) with: result toJson
```

Deux choses sont intéressantes dans cette méthode. Il y a tout d'abord l'utilisation de la méthode `urlDecoded` qui est appliqué la chaîne de caractères contenant le titre recherché. Cette méthode permet la gestion des caractères spéciaux tels que l'espace ou les caractères accentués. Si vous cherchez un post ayant pour titre "La reproduction des hippocampes", le service REST recevra en fait la chaîne de caractère "La%20reproduction%20des%20hippocampes" et une recherche avec celle-ci ne fonctionnera pas car aucun titre de post ne coïncidera. Il faut donc nettoyer la chaîne de caractères en remplaçant les caractères spéciaux avant de lancer la recherche.

Un autre point important est la gestion des codes d'erreur HTTP. Lorsqu'un serveur HTTP répond à son client, il glisse dans l'entête de la réponse une valeur numérique qui fournit au client de précieuses informations sur le résultat attendu. Si la réponse contient le code 200, c'est que tout s'est correctement passé et qu'un résultat est fourni au client (c'est d'ailleurs la valeur par défaut dans `Seaside/Rest`). Mais parfois, un problème survient. Par exemple, la requête demande à accéder à une ressource qui n'existe pas. Dans ce cas, il est nécessaire de retourner un code 404 (Not Found) pour l'indiquer au client. Un code 500 va indiquer qu'une erreur d'exécution a été rencontrée par

le service. Vous trouverez la liste exhaustive des codes d'erreur sur la page décrivant le protocole HTTP (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>). Il est très important de les gérer correctement tant au niveau de votre service REST qu'au sein de votre client REST car c'est ce qui va permettre à la couche cliente de réagir à bon escient en fonction du résultat du traitement exécuté par le serveur.

Notre serveur web de recherche par titre est pratiquement terminé. Il nous reste maintenant à modifier le point d'entrée du service pour qu'il soit capable d'appeler le code métier associé.

```
TBRestfulFilter >> search: aTitle
  <get>
  <path: '/posts/search?title={aTitle}'>
  <produces: 'application/json'>

  TBRestServiceSearch new
    title: aTitle;
    applyServiceWithContext: self requestContext
```

9.9 Chercher selon une période

Une autre méthode intéressante pour lancer une recherche consiste à extraire l'ensemble des posts créés entre deux dates qui définissent ainsi une période. Pour cette raison, la méthode `searchDateFrom:to:` reçoit deux arguments qui sont également définis dans la syntaxe de l'URL.

```
TBRestfulFilter >> searchDateFrom: beginString to: endString
  <get>
  <path: '/posts/search?begin={beginString}&end={endString}'>
  <produces: 'application/json'>
```

La partie métier est implémentée au sein de l'objet `TBRestServiceSearchDate` héritant de `TBRestService`. Deux variables d'instance permettent de définir la date de début et la date de fin de la période de recherche.

```
TBRestService subclass: #TBRestServiceSearchDate
  instanceVariableNames: 'from to'
  classVariableNames: ''
  package: 'TinyBlog-Rest'

TBRestServiceSearchDate >> from
  ^from

TBRestServiceSearchDate >> from: anObject
  from := anObject

TBRestServiceSearchDate >> to
  ^to

TBRestServiceSearchDate >> to: anObject
  to := anObject
```

La méthode `execute` convertit les deux chaînes de caractères en instances de l'objet `Date` à l'aide de la méthode `fromString`. Elle lit l'ensemble des posts à l'aide de la méthode `allBlogPosts`, filtre les posts créés dans la période indiquée et retourne le résultat au format JSON.

```
TBRestServiceSearchDate >> execute
  | posts dateFrom dateTo |

  dateFrom := Date fromString: self from.
  dateTo := Date fromString: self to.

  posts := TBBlog current allBlogPosts
```

```

        select: [ :each | each date between: dateFrom and: dateTo ].

        posts do: [ :each | result add: (each asDictionary) ].
        self dataType: (WAMimeType applicationJson) with: result toJson

```

Il serait judicieux ici d'ajouter certaines vérifications. Les deux dates sont-elles dans un format correct ? La date de fin est-elle postérieure à celle de début ? Nous vous laissons implémenter ces améliorations et gérer correctement les codes d'erreur HTTP.

La dernière étape consiste à compléter la méthode `searchDateFrom:to:` afin d'instancier l'objet `TBRestServiceSearchDate` lorsque le service `searchDateFrom:to:` est invoqué.

```

TBRestfulFilter >> searchDateFrom: beginString to: endString
    <get>
    <path: '/posts/search?begin={beginString}&end={endString}'>
    <produces: 'application/json'>

    TBRestServiceSearchDate new
        from: beginString;
        to: endString;
        applyServiceWithContext: self requestContext

```

A l'aide d'une URL telle que `http://localhost:8080/TinyBlog/posts/search?begin=2017/1/1&end=2017/3/30`, vous pouvez tester votre nouveau service REST (bien évidemment, les dates doivent être adaptées en fonction du contenu de votre base de test).

9.10 Ajouter un post

Voyons maintenant comment ajouter un nouveau post à notre blog à l'aide de REST. Étant donné qu'il s'agit ici de la création d'une nouvelle ressource, nous devons utiliser le verbe POST pour décrire l'action. Le chemin sera la ressource désignant la liste des posts.

```

TBRestfulFilter >> addPost
    <post>
    <consumes: '*/json'>
    <path: '/posts'>

```

La description du service REST comporte la directive `<consumes:>` qui précise à Seaside qu'il doit accepter uniquement des requêtes clientes contenant des données au format JSON. Le client doit donc obligatoirement utiliser le paramètre `Content-Type: application/json` au sein de l'entête HTTP.

La couche métier est constituée par l'objet `TBRestServiceAddPost` qui hérite de la classe `TBRestService`.

```

TBRestService subclass: #TBRestServiceAddPost
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'TinyBlog-Rest'

```

Seule la méthode `execute` doit être implémentée. Elle lit le flux de données et le parse à l'aide de la méthode de classe `fromString:` de l'objet `NeoJSONReader`. Les données sont stockées dans un dictionnaire contenu dans la variable locale `post`. Il suffit ensuite d'instancier un `TBPost` et de le sauvegarder dans la base de données. Par sécurité, l'ensemble de ce processus est réalisé au sein d'une exception afin d'intercepter un problème d'exécution qui pourrait rendre instable le serveur. La dernière opération consiste à renvoyer au client un résultat vide mais aussi et surtout un code HTTP 200 (OK) signalant que le post a bien été créé. En cas d'erreur, c'est le message d'erreur 400 (BAD REQUEST) qui est retourné.

```

TBRestServiceAddPost >> execute
    | post |

```

```
[
    post := NeoJSONReader fromString: (self context request rawBody).
    TBBlog current writeBlogPost: (TBPost title: (post at: #title) text: (post at:
    #text) category: (post at: #category)).
] on: Error do: [ self context request badRequest ].

self dataType: (WAMimeType textPlain) with: ''
```

Il ne vous reste plus qu'à ajouter l'instanciation de `TBRestServiceAddPost` au sein de la déclaration du point d'entrée REST.

```
TBRestfulFilter >> addPost
    <post>
    <consumes: '*/json'>
    <path: '/posts'>

    TBRestServiceAddPost new
        applyServiceWithContext: self requestContext
```

En guide de travaux pratiques, il vous est possible d'améliorer la gestion d'erreur de la méthode `execute` afin de différencier une erreur au sein de la structure des données transmises au serveur, du format utilisé ou encore lors de l'étape d'ajout du post à la base de données. Un service REST complet se doit de fournir une information pertinente au client afin d'explicitier la cause du problème.

9.11 Amélioration possibles

Au fil de ce chapitre, vous avez implémenté les principales briques d'une API REST permettant de consulter et d'alimenter le contenu d'un moteur de blog. Il reste bien sur des évolutions possibles et nous vous encourageons à les implémenter. Voici quelques propositions qui constituent des améliorations pertinentes.

Modifier un post existant

La modification d'un post existant peut facilement être réalisée. Il vous suffit d'implémenter un service REST utilisant le verbe HTTP PUT et d'encoder votre post avec la même structure que celle utilisée pour la création d'un post (service `addPost`). L'exercice consiste ici à implémenter correctement la gestion des codes d'erreurs HTTP. De nombreux cas sont possibles.

* 200 (OK) ou 201 (CREATED) si l'opération a réussi, * 204 (NO CONTENT) si la requête ne contient pas de données, * 304 (NOT MODIFIED) si aucun changement ne doit être appliqué (le contenu du post est identique), * 400 (BAD REQUEST) si les données transmises par le client sont incorrectes, * 404 (NOT FOUND) si le post devant être modifié n'existe pas, * 500 (INTERNAL SERVER ERROR) si un problème survient lors de la création du post dans la base de données.

Supprimer un post

La suppression d'un post sera le résultat d'une requête DELETE transmise au serveur. Ici aussi, il vous est conseillé d'implémenter une gestion la plus complète possible des codes d'erreurs HTTP qui devrait être assez proche de celle utilisée dans le service de modification d'un post.

Maîtriser les feuilles de styles avec RenoirSt

Les feuilles de styles (Cascading Style Sheet) sont devenues un élément incontournable du web. Elles permettent de séparer le fond de la forme en fournissant de puissants outils destinés à gérer l'apparence d'une page HTML. Grâce à elles, il n'est plus utile d'insérer des attributs définissant l'apparence d'un élément au sein même des balises HTML. L'ensemble de ces attributs est déporté au sein d'une ou plusieurs feuilles de styles attachées ou intégrées au document HTML. Il est ainsi aisé de modifier l'apparence des titres, de choisir la justification d'un paragraphe ou encore la couleur des boutons.

Pharo dispose d'un framework dédié à la définition de feuilles de styles. Il s'agit de RenoirST. Grâce à lui, vous allez définir les attributs CSS mais sans jamais manipuler les instructions CSS. Vous n'utiliserez donc que Pharo.

Notez que l'utilisation de RenoirST est optionnelle et que vous pouvez aussi utiliser des fichiers CSS de manière plus traditionnelle.

10.1 Installation de RenoirSt

Le framework RenoirSt est disponible dans le catalogue Pharo. Il vous suffit donc d'ouvrir le Catalog Browser et de sélectionner RenoirSt. Pour développer avec Seaside, il est recommandé d'installer également RenoirStPlusSeaside qui comprend RenoirSt ainsi que le code nécessaire à une intégration correcte au serveur d'applications de Pharo.

10.2 Principe de fonctionnement

RenoirSt est un générateur de code CSS. L'exécution d'une section de code Pharo déclenche la construction de la section CSS équivalente. Le résultat obtenu est une chaîne de caractères.

Le principe de fonctionnement est assez simple. Il vous suffit d'utiliser la méthode d'instance `declareRuleSetFor:with:` de l'objet `CascadingStyleSheetBuilder`. Le premier argument spécifie l'élément sur lequel le style est appliqué. Le second définit son apparence.

L'exemple suivant définit l'apparence d'un paragraphe de la classe `'resume'`.

```
CascadingStyleSheetBuilder new
  declareRuleSetFor: [ :selector | selector paragraph class: 'resume' ]
  with: [ :style | style color: CssSVGColors black ];
  build
```

Le résultat obtenu sera:

```
p.resume
{
  color: black;
}
```

Utiliser Pharo pour construire la feuille CSS de votre application permet évidemment de la générer dynamiquement. Une feuille de style est un document statique dans lequel il vous est nécessaire d'anticiper vos besoins. RenoirSt est l'outil parfait pour générer la présentation de votre application et l'adapter selon les situations rencontrées par votre application durant son fonctionnement.

10.3 Intégrer une feuille de style dans TinyBlog

Vous allez maintenant améliorer l'apparence visuelle de TinyBlog en modifiant la partie publique du logiciel.

Pour éviter que la feuille de style soit reconstruite systématiquement à chaque appel de la méthode `style` de `TBApplicationRootComponent`, nous allons utiliser une variable d'instance afin de la mettre en cache. Pour cela, ajoutez la variable d'instance `css` à `TBApplicationRootComponent`.

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

La méthode `TBApplicationRootComponent >> styleSheet` contiendra la définition de la feuille de styles et est appelée par la méthode `initialize`. Pour l'instant, contentez vous de retourner une chaîne vide.

```
TBApplicationRootComponent >> styleSheet
^''

TBApplicationRootComponent >> initialize
  super initialize.
  css := self styleSheet.
  main := TBPostsListComponent new.
```

A partir de maintenant, vous avez la possibilité de modifier l'apparence de TinyBlog.

10.4 Améliorer le visuel

Jusqu'ici, TinyBlog n'a pas été spécialement soigné visuellement. Il utilise le style de base fourni par le framework Bootstrap et c'est vrai, il n'est pas toujours très sexy. Grâce à la feuille de styles construite à l'aide de RenoirSt, vous allez lui donner un look plus travaillé.

Commençons par l'affichage des billets. Celui-ci a été défini dans `TBPostComponent` selon une très mauvaise méthode puisque l'apparence du titre et du sous-titre sont uniquement définis par les balises HTML `<h2>` et `<h6>` et qu'aucune classe CSS n'a été spécifiée. Corrigez le code à l'aide de cette nouvelle version de la méthode `renderContentOn::`

```
TBPostComponent >> renderContentOn: html
  html paragraph class: 'title'; with: self title.
  html paragraph class: 'subtitle'; with: self date.
  html paragraph class: 'content'; with: self text
```

Vous pouvez modifier le style de ces éléments dans la méthode `stylesheet`. Commencez par indiquer la taille du titre. Elle est fixée à 200% de la taille par défaut.

```
TBApplicationRootComponent >> styleSheet
  ^(CascadingStyleSheetBuilder new
    declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
```



```

with: [ :style | style fontSize: 200 percent ];
build) asString

```

Le sous-titre est écrit en italique et le contenu est justifié.

```

TBApplicationRootComponent >> styleSheet
^(CascadingStyleSheetBuilder new
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style | style fontSize: 200 percent ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'subtitle' ]
  with: [ :style | style fontStyle: CssConstants italic ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'content' ]
  with: [ :style | style textAlign: CssConstants justify ];
  build) asString

```

Le titre peut être amélioré en faisant en sorte que le premier caractère soit mis en valeur. Il sera plus grand et obligatoirement écrit à l'aide d'une lettre majuscule. Le titre est souligné à l'aide d'un trait fin comme afficher dans la figure 10-1.

```

TBApplicationRootComponent >> styleSheet
^(CascadingStyleSheetBuilder new
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style | style fontSize: 200 percent ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'subtitle' ]
  with: [ :style | style fontStyle: CssConstants italic ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'content' ]
  with: [ :style | style textAlign: CssConstants justify ];
  declareRuleSetFor: [ :selector | (selector paragraph class: 'title') firstLetter ]
  with: [ :style |
    style
      fontSize: 150 percent;
      textTransform: CssConstants capitalize
    ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style | style borderBottomStyle: CssConstants solid ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style | style borderWidth: 1px ];
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style | style borderColor: CssSVGColors black ];
  build) asString

```

10.5 Faciliter la maintenance

A ce stade, il est important de se poser la question de la maintenance de ce code. Il est évident que nous pouvons le refactoriser en regroupant la définition des attributs.

```

TBApplicationRootComponent >> styleSheet
^(CascadingStyleSheetBuilder new
  declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
  with: [ :style |
    style
      fontSize: 200 percent;
      borderBottomStyle: CssConstants solid;
      borderWidth: 1px;
      borderColor: CssSVGColors black
    ];
  declareRuleSetFor: [ :selector | (selector paragraph class: 'title') firstLetter ]
  with: [ :style |
    style
      fontSize: 150 percent;
      textTransform: CssConstants capitalize
    ];
  build) asString

```

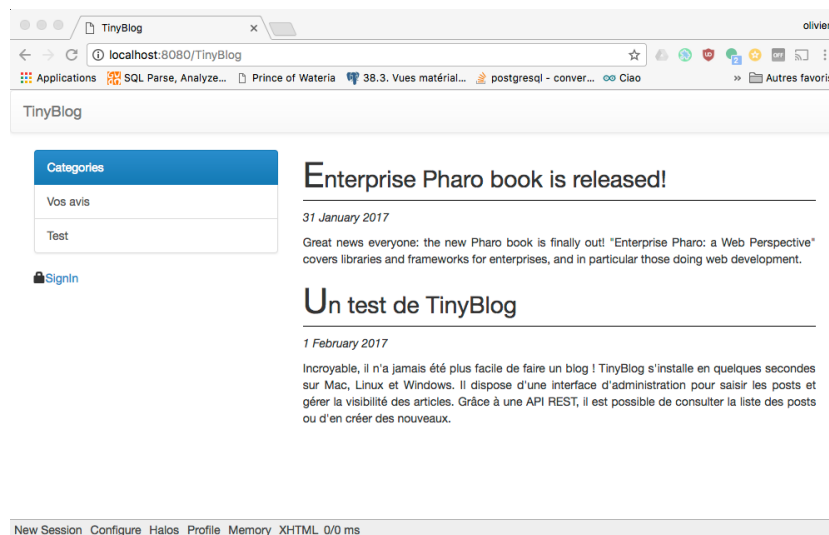


Figure 10-1 Amélioration de l'apparence de TinyBlog

```
];
declareRuleSetFor: [ :selector | selector paragraph class: 'subtitle' ]
  with: [ :style | style fontStyle: CssConstants italic ];
declareRuleSetFor: [ :selector | selector paragraph class: 'content' ]
  with: [ :style | style textAlign: CssConstants justify ];
build) asString
```

Commencez par extraire les définitions des différents styles et transformez les ensuite en méthodes. Vous pouvez les classer dans un protocole styles.

```
TBApplicationRootComponent >> titleStyleOn: aStyle.
  aStyle
    fontSize: 200 percent;
    borderBottomStyle: CssConstants solid;
    borderWidth: 1px;
    borderColor: CssSVGColors black;
    yourself

TBApplicationRootComponent >> titleFirstLetterStyleOn: aStyle.
  aStyle
    fontSize: 150 percent;
    textTransform: CssConstants capitalize;
    yourself

TBApplicationRootComponent >> paragraphSubtitleStyleOn: aStyle
  aStyle
    fontStyle: CssConstants italic;
    yourself

TBApplicationRootComponent >> paragraphContentStyleOn: aStyle
  aStyle
    textAlign: CssConstants justify;
    yourself
```

Vous pouvez donc simplifier la méthode `styleSheet` qui devient:

```
TBApplicationRootComponent >> styleSheet
  ^ (CascadingStyleSheetBuilder new
    declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
```

```

        with: [ :style | self titleStyleOn: style ];
declareRuleSetFor: [ :selector | (selector paragraph class: 'title') firstLetter ]
        with: [ :style | self titleFirstLetterStyleOn: style ];
declareRuleSetFor: [ :selector | selector paragraph class: 'subtitle' ]
        with: [ :style | self paragraphSubtitleStyleOn: style ];
declareRuleSetFor: [ :selector | selector paragraph class: 'content' ]
        with: [ :style | self paragraphContentStyleOn: style ];
build) asString

```

Poursuivez votre travail de refactorisation en extrayant les méthodes chargées d'appliquer les styles définis.

```

TBApplicationRootComponent >> applyTitleStyleOn: aSheet
    ^aSheet
        declareRuleSetFor: [ :selector | selector paragraph class: 'title' ]
            with: [ :style | self titleStyleOn: style ];
        declareRuleSetFor: [ :selector | (selector paragraph class: 'title') firstLetter ]
            with: [ :style | self titleFirstLetterStyleOn: style ]

TBApplicationRootComponent >> applyParagraphSubtitleStyleOn: aSheet
    ^aSheet declareRuleSetFor: [ :selector | selector paragraph class: 'subtitle' ]
        with: [ :style | self paragraphSubtitleStyleOn: style ]

TBApplicationRootComponent >> applyParagraphContentStyleOn: aSheet
    ^aSheet declareRuleSetFor: [ :selector | selector paragraph class: 'content' ]
        with: [ :style | self paragraphContentStyleOn: style ]

```

Il ne vous reste plus qu'à modifier la méthode `styleSheet` pour qu'elle prennent en compte ces nouvelles méthodes.

```

TBApplicationRootComponent >> styleSheet
    | styles |
    styles := CascadingStyleSheetBuilder new.
    self
        applyTitleStyleOn: styles;
        applyParagraphSubtitleStyleOn: styles;
        applyParagraphContentStyleOn: styles.
    ^styles build asString

```

Le résultat est beaucoup plus lisible et surtout bien plus facile à maintenir. Des styles peut être définis et utilisés sur différents éléments de la feuilles de styles.

10.6 Conclusion

L'utilisation de feuilles de styles autorise la modification profonde de l'aspect visuel de TinyBlog. A l'aide de `RenoirSt`, il ne vous est pas utile de connaître les détails de la syntaxe des CSS et vous utilisez uniquement la langage `Pharo` pour développer votre application. Par contre, une connaissance des mécanismes et du fonctionnement des CSS est requise pour exploiter correctement l'ensemble des possibilités de `RenoirSt`.

Utiliser des modèles de mise en page avec Mustache

Poursuivons l'amélioration de TinyBlog en nous intéressant à l'utilisation de modèles pour l'affichage des données de TinyBlog. Les modèles sont particulièrement utiles pour le développement web car ils permettent de mixer aisément des chaînes de caractères avec des balises HTML et ceci, sans avoir recours à de toujours pénibles manipulations des chaînes de caractères. Ils facilitent également la lecture et la maintenance du code. Ils sont très utiles pour la localisation des applications. Si votre logiciel doit être traduit en plusieurs langues, vous apprécierez forcément les modèles.

Parmi les technologies web, il existe de nombreux moteurs de gestion des modèles. Avec Pharo, vous disposez de l'adaptation de Mustache (<https://mustache.github.io/>) qui est un produit reconnu. L'installation est rapide puisque le framework Mustache est disponible dans le catalogue Pharo.

11.1 Ajouter un bas de page

L'objectif est d'ajouter un bas de page à l'écran principal de TinyBlog. Cette zone est généralement destinée à recevoir les mentions légales d'un site, des liens, les noms des auteurs et de nombreuses autres informations.

Pour mettre en place le bas de page, il nous faut tout d'abord ajouter la méthode `renderFooterOn:` chargée de l'affichage du bas de page. Vous devez bien évidemment modifier la méthode `renderContentOn:` de la classe `TBPostListComponent` en ajoutant l'appel à la méthode `renderFooterOn:`.

```
TBPostListComponent >> renderFooterOn: html
    html div class: 'footer'; with: [
        html text: 'I''am the footer!'
    ]

TBPostListComponent >> renderContentOn: html
    super renderContentOn: html.
    html render: (TBAuthenticationComponent from: self).
    html
        tbsContainer: [
            html tbsRow
                showGrid;
                with: [
                    self renderCategoryColumnOn: html.
                    self renderPostColumnOn: html
```

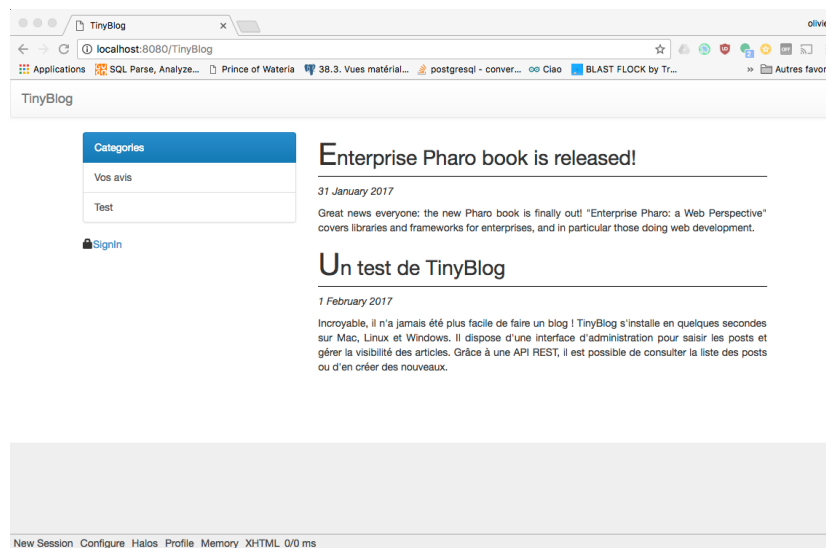


Figure 11-1 Le bas de page

```
].
  self renderFooterOn: html
]
```

Le style de la div contenant le bas de page est modifié à l'aide d'une classe CSS nommée `footer`. Celle ci nous permet de spécifier le style d'affichage du bas de page. Pour cela, ajoutez un appel à la méthode `applyFooterStyleOn:` à la déclaration de la feuille de styles.

```
TBApplicationRootComponent >> styleSheet
| styles |
styles := CascadingStyleSheetBuilder new.
self
  applyTitleStyleOn: styles;
  applyParagraphSubtitleStyleOn: styles;
  applyParagraphContentStyleOn: styles;
  applyFooterStyleOn: styles.
^styles build asString
```

Cette nouvelle méthode sélectionne les éléments de la classe `footer` et applique sur eux le style défini par une autre méthode nommée `footerStyleOn:`.

```
TBApplicationRootComponent >> applyFooterStyleOn: aSheet
^aSheet declareRuleSetFor: [ :selector | selector div class: 'footer' ]
with: [ :style | self footerStyleOn: style ]
```

La méthode `footerStyleOn:` fixe les attributs CSS afin de placer la div en bas de la page. Un fond gris est appliqué et le texte est centré.

```
TBApplicationRootComponent >> footerStyleOn: aStyle
aStyle
  position: CssConstants absolute;
  bottom: 0 pixels;
  paddingTop: 25 pixels;
  height: 150 pixels;
  width: 100 percent;
  backgroundColor: (CssRGBColor red: 239 green: 239 blue: 239);
  textAlign: CssConstants center;
  yourself
```

Vous pouvez maintenant ajouter du contenu et pour cela, vous allez utiliser Mustache.

11.2 Ajouter du contenu dans le bas de page

Pour ajouter des éléments dans le bas de page, vous allez utiliser certaines fonctionnalités de Mustache qui facilitent grandement la substitution d'éléments au sein de modèles. Avec Mustache, il est aisé de manipuler des éléments textuels statiques mais également des textes générés dynamiquement.

11.3 Utiliser de texte statique

Le premier modèle doit afficher les principales technologies utilisées dans TinyBlog. Pour cela, définissez une méthode `renderPoweredByOn` dans la classe `TBPostListComponent`. Un dictionnaire contient les données qui sont insérées au sein du modèle. Celui-ci est défini par une chaîne de caractères dans laquelle les éléments devant être substitués sont encadrés par les caractères `"{{"` et `"}}"`.

Par défaut, Mustache utilise les caractères spéciaux d'HTML pour assurer un rendu web optimal (par exemple `Pharo` est transformé en `Pharo`). Si vous ne voulez pas les utiliser, vous devez encadrer les éléments par les caractères `{{ et }}`.

```
TBPostListComponent >> renderPoweredByOn: html
  html text: ('Powered by {{language}}, {{framework}} and {{tool}}.' asMustacheTemplate
    value: {
      'language' -> 'Pharo'.
      'framework' -> 'Seaside'.
      'tool' -> 'Bootstrap'
    } asDictionary)
```

Vous pouvez maintenant modifier la méthode `renderFooterOn` afin d'afficher le texte sur la page.

```
TBPostListComponent >> renderFooterOn: html
  html div class: 'footer'; with: [
    self renderPoweredByOn: html.
  ]
```

11.4 Utiliser du texte généré dynamiquement

Avec Mustache, il est également possible de remplacer des éléments au sein d'un modèle à l'aide d'un texte généré dynamiquement. Ici par exemple, la méthode `renderDateTodayOn` permet à TinyBlog de construire un texte contenant la date du jour. Le code exécuté doit être placé entre crochets au sein du dictionnaire définissant les données à insérer.

```
TBPostListComponent >> renderDateTodayOn: html
  html text: ('The date today is {{today}}.' asMustacheTemplate value: { 'today' -> [
    Date today ] } asDictionary)
```

Pour que la date apparaisse sur la page, il vous faut ajouter l'appel à la méthode `renderFooterOn`.

```
TBPostListComponent >> renderFooterOn: html
  html div class: 'footer'; with: [
    self renderDateTodayOn: html.
    html break.
    self renderPoweredByOn: html.
  ]
```

11.5 Conclusion

Nous n'avons ici qu'effleurer le potentiel de Mustache. Il simplifie réellement la construction d'éléments au sein d'une application web. Ce framework propose de nombreuses autres fonctionnalités.

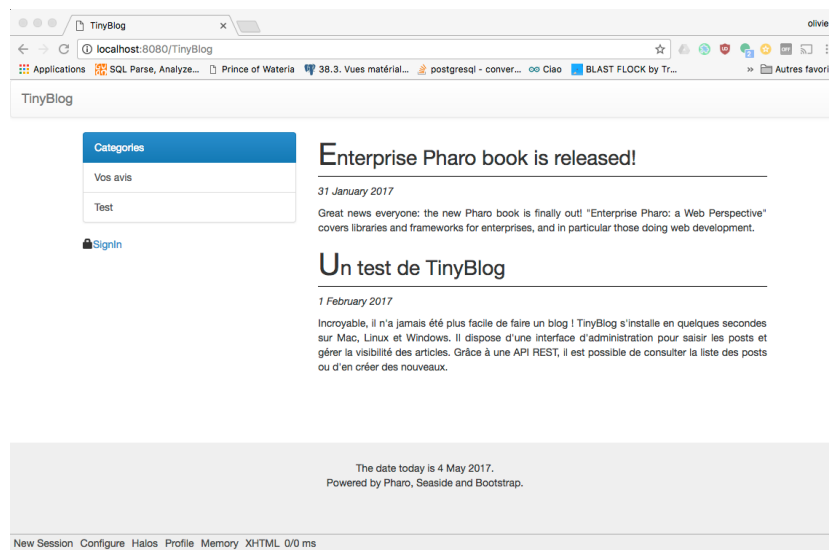


Figure 11-2 Le bas de page

Pour en savoir plus et explorer toutes ses possibilités, nous vous invitons à consulter le chapitre qui lui est consacré au sein du livre "Enterprise Pharo".

Exportation de données

Tout bon logiciel se doit de disposer de fonctionnalités permettant l'exportation des données qu'il manipule. Dans le cadre de TinyBlog, il est ainsi intéressant de proposer à l'utilisateur d'exporter en PDF un post afin d'en conserver la trace. Il pourra également l'imprimer aisément avec une mise en page adaptée. Pour l'administrateur du blog, il est utile de proposer des fonctionnalités d'exportation en CSV et en XML afin de faciliter la sauvegarde du contenu du blog. En cas d'altération de la base de données, l'administrateur dispose alors d'une solution de secours pour remettre en ordre son instance de l'application dans les plus brefs délais. Proposer des fonctionnalités d'exportation permet également d'assouplir l'utilisation d'un logiciel en favorisant l'interopérabilité, c'est à dire l'échange des données avec d'autres logiciels. Il n'y a rien de pire qu'un logiciel fermé ne sachant communiquer avec personne.

12.1 Exporter un article en PDF

Le format PDF (Portable Document Format) a été créé par la société Adobe en 1992. C'est un langage de description de pages permettant de spécifier la mise en forme d'un document ainsi que son contenu. Il est particulièrement utile pour concevoir des documents électroniques, des eBooks et dans le cadre de l'impression puisqu'un document PDF conserve sa mise en forme lorsqu'il est imprimé. Vous allez justement mettre à profit cette propriété en ajoutant à TinyBlog la possibilité d'exporter un post sous la forme d'un fichier PDF.

Artefact

La construction d'un document PDF avec Pharo est grandement simplifiée à l'aide d'un framework nommé Artefact (<https://sites.google.com/site/artefactpdf/>). Pour l'installer, il vous suffit de le sélectionner dans le catalogue Pharo.

Intégrer l'exportation dans la liste des posts

Pour pouvoir exporter un post en PDF, l'utilisateur doit disposer d'un lien sur chaque post. Pour cela, vous devez modifier la méthode `TBPostComponent >> renderContentOn:`.

```
renderContentOn: html
  html paragraph class: 'title'; with: self title.
  html paragraph class: 'subtitle'; with: self date.
  html paragraph class: 'content'; with: self text.
  html anchor
    callback: [ TBPostPDFExport context: self requestContext post: post ];
    with: 'PDF'
```

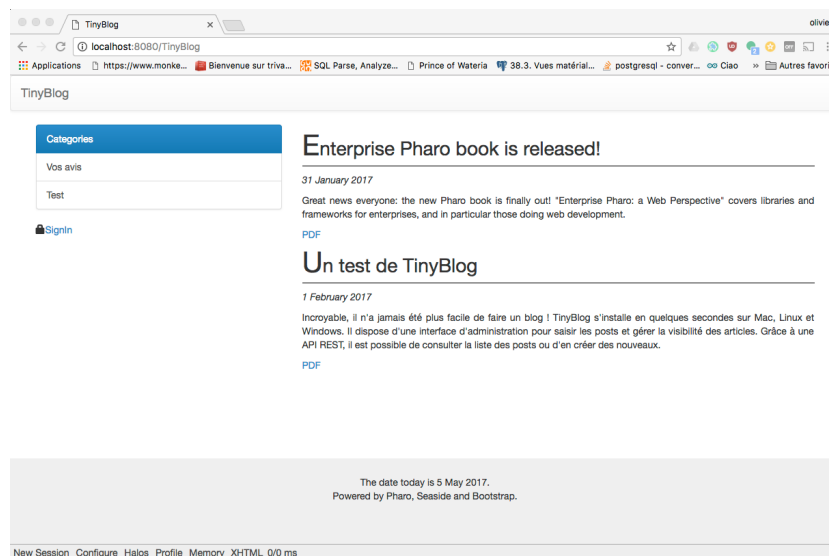


Figure 12-1 Chaque post peut être exporté en PDF

Lorsque l'utilisateur clique sur le lien, une instance de la classe `TBPostPDFExport` est créée. Cette classe aura la responsabilité de construire et d'envoyer le document PDF à l'utilisateur. C'est pour cette raison qu'elle a besoin de deux paramètres qui sont le contexte HTTP et le post sélectionné.

Construction du document PDF

Vous allez maintenant implémenter la classe `TBPostPDFExport`. Celle-ci nécessite deux variables d'instance qui sont `post` contenant le post sélectionné et `pdfdo` pour stocker le document PDF généré.

```
Object subclass: #TBPostPDFExport
  instanceVariableNames: 'post pdfdoc'
  classVariableNames: ''
  category: 'TinyBlog-Export'

TBPostPDFExport >> post
  ^ post

TBPostPDFExport >> post: aPost
  post := aPost
```

Vous avez besoin de la méthode de classe `context:post:` qui est le point d'entrée pour utiliser la classe. Celle-ci appelle la méthode `renderPDFFile` qui produit le document PDF.

```
TBPostPDFExport class >> context: anHTTPContext post: aPost
  ^ self new
    post: aPost;
    renderPDFFile;
    yourself
```

Envoi du document au client

Check with olivier

12.2 Exportation des posts au format CSV

Vous allez poursuivre l'amélioration de TinyBlog en ajoutant une option dans la partie "Administration" de l'application. Celle-ci doit permettre l'exportation de l'ensemble des billets du blog dans un fichier CSV. Ce format (Comma-separated values) est un format bien connu des utilisateurs de tableurs qui l'exploitent souvent pour importer ou exporter des données. Il s'agit d'un fichier texte dans lequel les données sont formatées et distinctes les unes des autres à l'aide d'un caractère séparateur qui est le plus souvent une virgule. Le fichier est donc composé de lignes et chacune d'entre elles contient un nombre identique de colonnes. Une ligne se termine par un caractère de fin de ligne (CRLF).

Pour gérer le format CSV dans Pharo, vous disposez du framework NeoCSV installable à l'aide du catalogue.

12.3 Ajouter l'option d'exportation

L'utilisateur doit disposer d'un lien pour déclencher l'exportation des billets au format CSV. Ce lien est ajouté sur la page d'administration, juste en dessous du tableau référençant les billets publiés. Vous devez donc éditer la méthode `TBPostsReport>>renderContentOn:` afin d'ajouter une ancre et un callback.

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

  super renderContentOn: html.

  html tbsGlyphIcon perform: #iconCloudDownload.
  html anchor
    callback: [ self exportToCSV ];
    with: 'Export to CSV'.
```

Cette méthode devient un peu trop longue. Il est temps de la fragmenter et d'isoler les différents éléments composant l'interface utilisateur.

```
TBPostsReport>>renderAddPostAnchor: html
  html tbsGlyphIcon perform: #iconPencil.
  html anchor
    callback: [ self addPost ];
    with: 'Add post'

TBPostsReport>>renderExportToCSVAnchor: html
  html tbsGlyphIcon perform: #iconCloudDownload.
  html anchor
    callback: [ self exportToCSV ];
    with: 'Export to CSV'

TBPostsReport>>renderContentOn: html
  self renderAddPostAnchor: html.
  super renderContentOn: html.
  self renderExportToCSVAnchor: html
```

Il vous faut maintenant implémenter la méthode `TBPostsReport>>exportToCSV`. Celle-ci génère une instance de la classe `TBPostsCSVExport`. Cette classe doit transmettre au client un fichier CSV et doit donc connaître le contexte HTTP afin de pouvoir répondre. Il faut également lui transmettre le blog à exporter.

```
TBPostsReportexportToCSV
  TBPostsCSVExport context: self requestContext blog: self blog
```

12.4 Implémentation de la classe TBPostsCSVExport

La méthode de classe `context:blog: initialize` une instance de `TBPostsCSVExport` et appelle la méthode `TBPostsCSVExport>>sendPostsToCSVFrom:to:.`

```
Object subclass: #TBPostsCSVExport
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'TinyBlog-Export'

TBPostsCSVExport class >> context: anHTTPContext blog: aBlog
  ^ self new
    sendPostsToCSVFrom: aBlog to: anHTTPContext
    yourself
```

Cette méthode lit le contenu de la base et génère grâce à NeoCSV le document CSV. La première étape consiste à déclarer un flux binaire qui sera par la suite transmis au client.

```
TBPostsCSVExport >> sendPostsToCSVFrom: aBlog to: anHTTPContext
  | outputStream |

  outputStream := (MultiByteBinaryOrTextStream on: (OrderedCollection new)) binary.
```

La partie importante de la méthode utilise NeoCSV pour insérer dans le flux de sortie chaque billet converti au format CSV. Le titre, la date de publication et le contenu du billet sont séparés par une virgule. Lorsque cela est nécessaire (titre et contenu), NeoCSV utilise des guillemets pour indiquer que la donnée est une chaîne de caractères. La méthode `nextPut:` permet d'insérer au début du fichier les noms des colonnes. La méthode `addObjectFields:` sélectionne les données ajoutées au fichier et récoltées à l'aide de la méthode `allBlogPosts`.

```
  outputStream nextPutAll: (String streamContents: [ :stream |
    (NeoCSVWriter on: stream)
      nextPut: #('Title' 'Date' 'Content');
      addObjectFields: {
        [ :post | post title ].
        [ :post | post date ].
        [ :post | post text ] };
      nextPutAll: (aBlog allBlogPosts)
    ]).
```

Il ne vous reste plus qu'à transmettre les données au navigateur du poste client. Pour cela, il vous faut produire une réponse dans le contexte HTTP de la requête. Le type MIME (`text/csv`) et l'encodage (`UTF-8`) sont déclarés au navigateur. La méthode `attachmentWithFileName:` permet de spécifier un nom de fichier au navigateur.

```
  anHTTPContext respond: [:response |
    response
      contentType: 'text/csv; charset=UTF-8';
      attachmentWithFileName: 'posts.xml';
      binary;
      nextPutAll: (outputStream reset contents)
  ]
```

12.5 Exportation des posts au format XML

XML est un autre format populaire pour exporter des informations. Ajouter cette fonctionnalité à TinyBlog ne sera pas difficile car Pharo dispose d'un excellent support du format XML. Pour in-

staller le framework permettant de générer du XML, sélectionnez XMLWriter dans le catalogue Pharo. Les classes sont regroupées dans le paquet XML-Writer-Core.

Mise à jour de l'interface utilisateur

Vous allez ajouter une fonctionnalité afin d'exporter dans un fichier XML l'ensemble des billets contenus dans la base. Il faut donc ajouter un lien sur la page d'administration.

```
TBPostsReport >> renderExportToXMLAnchor: html
  html tbsGlyphIcon perform: #iconCloudDownload.
  html anchor
    callback: [ self exportToXML ];
    with: 'Export to XML'
```

```
TBPostsReport >> renderContentOn: html
  self renderAddPostAnchor: html.
  super renderContentOn: html.
  self renderExportToCSVAnchor: html.
  self renderExportToXMLAnchor: html
```

Factorisons le code pour regrouper les deux fonctionnalités d'exportation au sein d'une seule méthode. Un caractère séparateur sera également judicieux pour améliorer l'affichage en évitant que les deux liens ne soient collés l'un à l'autre.

```
TBPostsReport >> renderExportOptionsOn: html
  self renderExportToCSVAnchor: html.
  html text: ' '.
  self renderExportToXMLAnchor: html
```

```
TBPostsReport >> renderContentOn: html
  self renderAddPostAnchor: html.
  super renderContentOn: html.
  self renderExportOptionsOn: html
```

12.6 Génération des données XML

La nouvelle méthode exportToXML instancie l'objet TBPostsXMLExport qui a la responsabilité de générer le document XML.

```
TBPostsReport >> exportToXML
  TBPostsXMLExport context: self requestContext blog: self blog
```

Il vous faut maintenant implémenter la classe TBPostsXMLExport. Celle-ci contient une méthode de classe context:blog: qui reçoit le contexte de la requête HTTP et la liste des billets.

```
Object subclass: #TBPostsXMLExport
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'TinyBlog-Export'

TBPostsXMLExport class >> context: anHTTPContext blog: aBlog
  ^ self new
    sendPostsToXMLFrom: aBlog to: anHTTPContext
    yourself
```

La méthode d'instance sendPostsToXMLFrom:to: prend en charge la conversion des données contenues dans les instances de TBPPost vers le format XML. Pour cela, vous avez besoin d'instancier la classe XMLWriter et de sauvegarder l'instance dans la variable locale xml. Celle-ci contiendra le fichier XML produit.

```
TBPostsXMLExport >> sendPostsToXMLFrom: aBlog to: anHTTPContext
| xml |

xml := XMLWriter new enablePrettyPrinting.
```

La message `enablePrettyPrinting` modifie le comportement du générateur XML en forçant l'insertion de retour à la ligne entre les différentes balises. Ceci facilite la lecture d'un fichier XML par un être humain. Si le document généré est volumineux, ne pas utiliser cette option permet de réduire la taille des données.

Vous pouvez maintenant formater les données en XML. La message `xml` permet d'insérer un entête au tout début des données. Chaque billet est placé au sein d'une balise `post` et l'ensemble des billets est stocké au sein de la balise `posts`. Pour celle ci, un espace de nommage `TinyBlog` est défini et pointe sur le domaine `pharo.org`. Chaque balise `post` est définie au sein du parcours de la collection retournée par la méthode `allBlogPosts`. Le titre est conservé tel quel, par contre la date est convertie au format anglosaxon (`year-month-day`). Notez le traitement particulier appliqué sur le texte du billet. Celui ci est encadré par une section `CDATA` afin de gérer correctement les caractères spéciaux pouvant s'y trouver (retour à la ligne, lettres accentuées, etc.).

```
xml writeWith: [ :writer |
  writer xml.
  writer tag
    name: 'posts';
    xmlnsAt: 'TinyBlog' put: 'www.pharo.org/tinyblog';
    with: [
      aBlog allBlogPosts do: [ :post |
        writer tag: 'post' with: [
          writer tag: 'title' with: post title.
          writer tag: 'date' with: (post date yyyyymmdd).
          writer tag: 'text' with: [ writer cdata: post text ].
        ]
      ]
    ].
].
```

La dernière étape consiste à retourner le document XML au client. Le type MIME utilisé ici est `text/xml`. Le fichier généré porte le nom de `posts.xml`.

```
anHTTPContext respond: [:response |
  response
    contentType: 'application/xml; charset=UTF-8';
    attachmentWithFileName: 'posts.xml';
    nextPutAll: (xml contents)
].
```

Quelques dizaines de lignes de code ont permis d'implémenter l'exportation en XML des billets. Votre moteur de blog dispose maintenant de fonctionnalités d'exportation et d'archivage des données.

12.7 Amélioration possibles

Il existe de nombreux autres formats utiles pour l'exportation des données. Nous vous proposons d'ajouter le format JSON à la boîte à outils de `TinyBlog`. Pour cela, nous vous recommandons d'utiliser le framework `NeojSON` disponible dans le catalogue `Pharo`.

Une autre amélioration consiste à écrire un outil d'importation permettant de charger le contenu d'un fichier CSV ou XML dans la base de données de `TinyBlog`. Cette fonctionnalité vous permettra de restaurer le contenu de la base de données si un problème technique survient.

Charger le code des chapitres

13.1 Chapitre : Modèle

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant :

```
Metacello new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  version: #week1solution;
  configuration: 'TinyBlog';
  load
```

Vous pouvez ouvrir un browser de code sur la classe TBlog. Maintenant que vous avez la correction, vous pouvez compléter le code de votre application TinyBlog si nécessaire.

13.2 Chapitre : Extension du modèle et tests unitaires

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant:

```
Metacello new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
  version: #week2solution;
  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 TBlogTest 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 TBlogTest.

Ouvrez maintenant un browser de code pour regarder le code des classes TBlog et TBlogTest et compléter votre propre code si nécessaire. Avant de poursuivre, n'oubliez pas de committer une nouvelle version dans votre dépôt sur Smalltalkhub ou SS3 si vous avez modifié votre application.

13.3 Chapitre : Persistance des données de TinyBlog avec Voyage et Mongo

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant:

```
Metacello new
  smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
```

```
[
    version: #week4solution;
    configuration: 'TinyBlog';
    load
]
```

Pour tester le code, vous devez lancer le serveur HTTP pour Seaside avec l'outil Seaside Control Panel (cf. sujet semaine précédente) ou avec le code suivant :

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

Vous devrez également faire :

```
[ TBBlog reset ;
  createDemoPosts
```

13.4 Chapitre : Construction d'une interface Web en Seaside pour TinyBlog

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant:

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

Puis, testez le code en exécutant:

```
[ TBTeapotWebApp start
```

Attention, vérifiez d'abord que votre propre serveur HTTP ne soit pas déjà lancé sur le même port . Si c'est le cas, arrêtez votre application ou changez son port.

Vous devez ajouter des posts dans le blog pour voir quelquechose.

```
[ TBBlog reset ; createDemoPosts
```

Avant de passer à la suite, stoppez votre serveur Teapot:

```
[ TBTeapotWebApp stop
```

13.5 Chapitre : Interface Web d'administration pour TinyBlog

Vous pouvez charger l'application complète TinyBlog en exécutant :

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

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

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

Si vous avez besoin de créer quelques posts initiaux:

```
[ TBBlog reset ; createDemoPosts
```

Complétez ou terminez **votre** application et commitez votre code dans votre dépôt sur Smalltalkhub.

13.6 Chapitre : Une Interface Web avec Teapot pour TinyBlog

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant:


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

Assurez vous que votre serveur Teapot est bien stoppé.

```
[ TBTeapotWebApp stop
```

